Implementing a Standards-based Distributed Measurement and Control Application on the Internet

Richard D. Schneeman
Computer Scientist
Sensor Integration Group
rschneeman@nist.gov
Telephone: (301) 975-4352

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
  and Technology
Manufacturing Engineering Laboratory
Automated Production Technology
  Division
Gaithersburg, MD 20899   USA

NIST

Implementing a Standards-based Distributed Measurement and Control Application on the Internet

Richard D. Schneeman
Computer Scientist
Sensor Integration Group
rschneeman@nist.gov
Telephone: (301) 975-4352

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
  and Technology
Manufacturing Engineering Laboratory
Automated Production Technology
  Division
Gaithersburg, MD 20899   USA

June 1999

Abstract

The distributed measurement and control (DMC) industry is migrating away from proprietary hardware and software platforms in favor of open and standardized approaches. High-level programming languages, object-oriented platforms, Internet technology, and standardized transducer interfaces are serving to shape the next-generation DMC landscape. This paper describes a framework for standardized DMC integration and a demonstration implementation developed at the National Institute of Standards and Technology (NIST). The framework details three key areas where standards will play critical roles in next generation DMC development, including (1) standardized transducer interfacing, (2) open network communications, and (3) distributed control applications. The demonstration implementation merges defacto networking standards from the Internet community with newly emerging smart transducer interface standards from the Institute of Electrical and Electronics Engineers (IEEE). Throughout the paper, design and implementation information using state-of-the-art hardware and software technologies for DMC deployment is described in terms of a real world Internet-based application environment.

iv

Table of Contents

Introduction

This paper describes a standardization framework and demonstration developed at NIST for implementing next generation distributed measurement and control (DMC) systems on the Internet. In its most basic form, a DMC system consists of transducers (i.e., sensors and actuators) connected together via a control network as shown in Figure 1. Sensors provide environmental input such as temperature or pressure to a measurement process. Actuators typically effect a physical change in system state. For example, in a thermostatically controlled setting, a fan might be turned on or off (actuated) to cool a room based on the temperature value collected by a sensor.

Control Network

| Smart Network Node | Smart Network Node | Smart Network Node |

Temperature Sensor   Pressure Sensor   Fan Actuator

Environmental Process
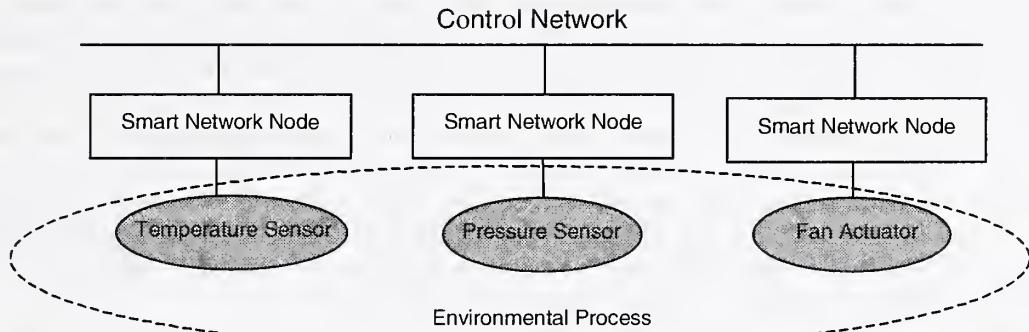
Figure 1: A basic distributed measurement and control system.

The transducers shown in Figure 1 transport information by connecting to a control network via smart network nodes. Smart network nodes are generally small microprocessor-based modules. The capability of the nodes to provide both autonomous and distributed intelligence to the system leads to the name smart network nodes. The smart network nodes provide the transducer with access to the network, and are responsible for processing data from both transducer devices and the network interface.

Figure 1 illustrates a basic view of a DMC system. A large-scale DMC system in a manufacturing setting today requires a great deal of engineering, time, and money to develop. Two factors adding to the complexity of building such systems are the lack of standards and the lack of off-the-shelf, high-level development environments. Many companies are searching for ways to cost-effectively implement DMC systems. Large manufacturing companies such as General Motors and Boeing have begun to re-evaluate how their institutional control networking technology can leverage standardized Internet-based networks and protocols [1]. Internet technologies such as the Java programming language, World Wide Web (WWW), Transmission Control Protocol/Internet Protocol (TCP/IP), and Ethernet are rapidly becoming the platforms of choice for building next generation DMC networks and applications. Additionally, DMC devices such as sensors and actuators are incorporating newly emerging Institute of Electrical and Electronics Engineers (IEEE) smart transducer interface standards. To further reduce costs, companies are beginning to shift their system and application development approaches to object-oriented, high-level programming language environments.

NIST researchers in the Manufacturing Engineering Laboratory (MEL) are exploring techniques to integrate Internet technologies and IEEE transducer interface standards with DMC. From this research, a standards-based framework for Internet DMC deployment has been identified. Key areas of the NIST DMC framework include (**1**) standardized transducer interfacing, (**2**) open network communications, and (**3**) distributed control applications. As part of this research, an Internet-based DMC demonstration project has been developed. The demonstration project provides the ability to access, control, and monitor a DMC system in real-time from the plant floor or from around the globe using the Internet.

The Need for DMC Standardization

When setting out to build a DMC system, engineers must address several key issues during the design stage. First, they must describe the types of transducers and other devices in the system as well as the microprocessor-based hardware interfaces required for each type of transducer. Second, they must take into account the availability of a proprietary control network for their problem domain. Finally, the algorithms and applications must be developed using a relevant distribution model. It is these three key areas of the DMC design phase that standardization efforts are underway to achieve more cost-effective DMC solutions.

In describing the types of transducers needed in the DMC system, engineers must first understand the dizzying array of methods used to interface transducers to microprocessor-based network nodes. In fact, it is this wide range of proprietary transducer interconnection strategies that has severely fragmented the transducer marketplace. This fact alone is driving the transducer market into adopting standardized interfaces for attaching transducers to both control networks as well as microprocessor-based hardware devices [2,3,4]. There are standards efforts currently underway within an IEEE committee that is addressing this transducer-interfacing problem [5,6]. The NIST framework uses the standards being developed by the IEEE to mitigate the multiplicity of interfaces found in today's fragmented transducer marketplace. This problem area is the first under consideration by the NIST Internet-based DMC framework in the Section titled *Standardized Transducer Interfacing*.

After selecting the appropriate transducers for the system, the control networking requirements must now be defined. Constraints on the network caused by transducer speed, application requirements, or hardware optimizations must be met by a particular networking medium or topology. In many cases, several types of control network mediums must be used because of the disparate capabilities imposed by different networks. This results in the specification of highly proprietary and complex network designs. However, the rapid transition to Ethernet-based networks as the new medium for control networking has reduced these network selection and integration problems [7,8,1,9,10,11]. Ethernet and Internet-based network protocols such as TCP/IP are being used as the common backbone for DMC-based networking strategies [12,13,14,15]. The NIST Internet-based DMC framework addresses the issues of developing a control network based on Internet technology in the Section titled *Open Network Communications*.

The network selected for the DMC design will certainly impact the type of network nodes used by it. The network nodes contain the control algorithms and applications that define the transducers purpose on the network. Implementations of control applications for DMC systems may use a centralized approach as seen in classic master/slave configurations, or they may be highly distributed or de-centralized in nature. IEEE standards are evolving to help make implementing transducer-based DMC applications in a distributed environment much more straightforward. The NIST Internet-based DMC framework provides standardization insight into the partitioning of distributed control among network nodes as well as other application design issues. The Section titled *Distributed Control Applications* addresses these topics.

The standardization issues addressed during the design phase of building a DMC system have given rise to an Internet-based DMC framework developed at NIST. The key areas of the NIST Internet-based DMC framework include (1) standardized transducer interfacing, (2) open network communications, and (3) distributed control applications. The NIST DMC framework addresses the design choices that face the DMC community in terms of standardization options. In order to illustrate the NIST DMC framework in terms of tangible hardware and software, NIST researchers developed a prototypical demonstration that encapsulates all aspects of the framework. An overview of the NIST DMC demonstration project is described in the next Section titled *The NIST Internet-based DMC Demonstration Project*.

The NIST Internet-based DMC Demonstration Project

Using the NIST Internet-based DMC framework as a design template, NIST researchers developed an Internet-based DMC demonstration that incorporates current standardization efforts. The DMC demonstration project addresses the problem of providing Internet access to remotely monitor and control an industrial process using a standardized framework.

The specific problem area addressed in the NIST DMC demonstration project is as follows. In the NIST machine shop a high-speed, precision machine tool is engaged in a long running job of milling large metal parts. During the milling process, a means of maintaining a relatively constant temperature of the material is required. Large temperature differentials in an open machine shop environment can result in dimensional variation of the finished products. A coolant tank shown in Figure 2 serves as the coolant source to the part being milled. An Internet-based DMC closed-loop control system was designed to regulate the temperature in the coolant tank.

Hardware components of the NIST DMC demonstration

Figure 2 illustrates the mechanical components of the DMC-based control system (control nodes are not shown in the figure). The goal is to keep the coolant in the tank to within a specified temperature range. During the machining process the coolant temperature tends to rise due to the heat generated in the cutting process. The increase in ambient air temperature around the machine also contributes to the rise in coolant temperature as well. A chiller maintains a reservoir of cold water that is constantly circulated. The coolant pumped from the tank is chilled in the chiller through a heat exchanger. The coolant from the chiller is either directed into the coolant tank or circulated back to the reservoir using a 3-way valve.
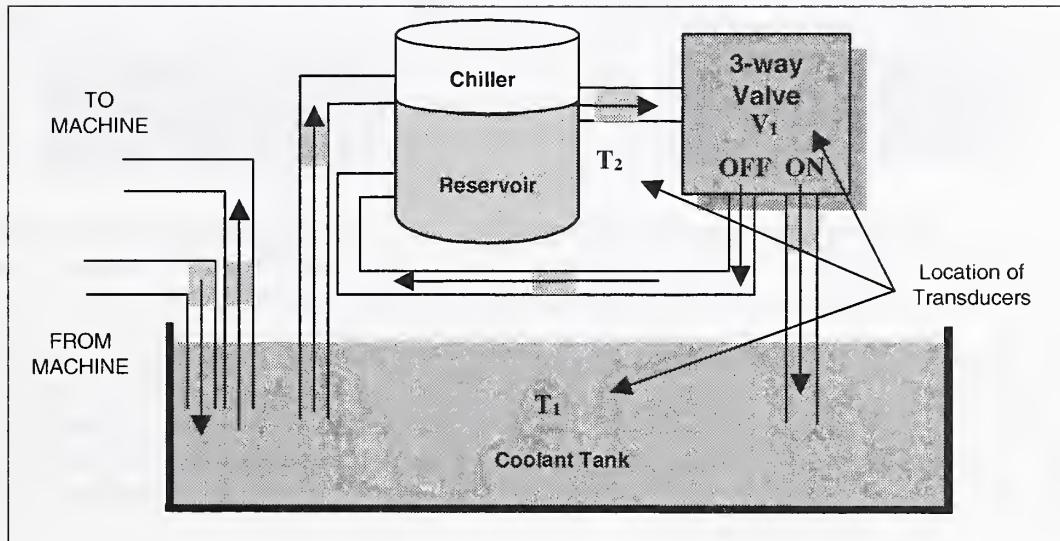
Figure 2: The mechanical parts of the NIST DMC control-loop application.

Figure 2 illustrates how this redirection process is done using a 3-way valve (V_1). The valve when turned OFF allows the coolant to flow back to the reservoir. When the valve has been turned ON or actuated, the chilled coolant flows into the coolant tank reducing the overall temperature. The temperatures in the demonstration are sensed with two temperature sensors located in the tank (T_1) and in the output line of the chiller (T_2). When the coolant temperature rises, the control-loop algorithm compensates for this and immediately signals the valve ON (actuate). When the temperature in the tank is reduced to a sufficient

level, the valve is turned OFF. This process is repeated with the valve mechanism oscillating ON and OFF to regulate the temperature. The result is a consistent coolant temperature (within 1/10 of a degree Fahrenheit) maintained throughout the parts machining cycle.

Software and network topology of the NIST DMC demonstration

The goal of the NIST Internet-based DMC demonstration project was to provide a means via standard Internet technology to access, view, monitor, and control this DMC application in real-time. In order to provide this capability, several key hardware and software technologies were developed. The overall NIST Internet-based DMC demonstration project topology is shown in Figure 3.
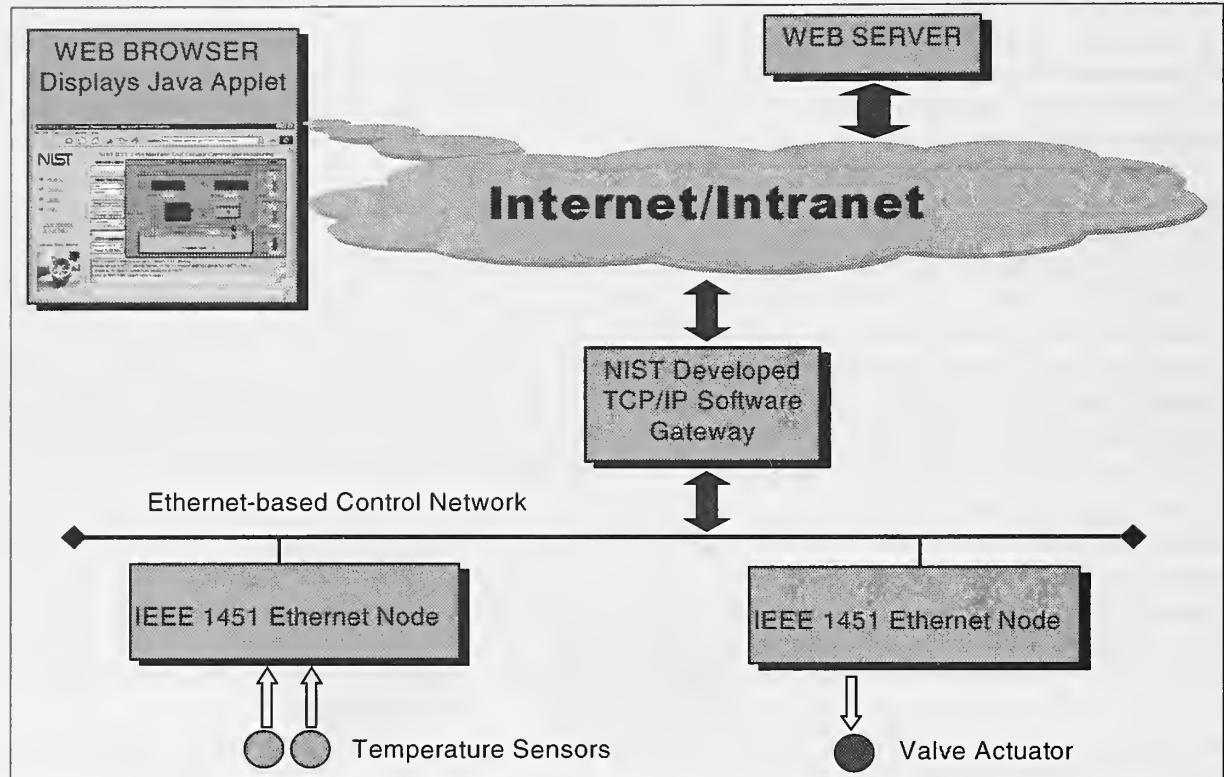
Figure 3: The overall NIST Internet-based DMC demonstration topology.

The Java applet shown in Figure 3 executes from within a Web browser and provides the information delivery mechanism to the shop floor or the operations manager's desk across the country. Any Web browser that supports Java can be used to view this application. A Web server provides the Java and hypertext markup language (HTML) data file repository capabilities for the browser in this demonstration. The actual DMC system was built using two state-of-the-art, Ethernet-based network node prototypes from Hewlett-Packard (HP)[1].

The HP network nodes support a prototype implementation of the IEEE P1451 specifications that provide (a) standardized application software interfaces to the transducer and the network and (b) standardized hardware interfaces for connecting transducers at the digital level to the Ethernet nodes. The temperature sensors and the valve actuator were directly controlled in this distributed setting using C language

---

[1] This document discusses concepts and architectural issues with early prototypes from Hewlett-Packard. The proprietary technology discussed here has been replaced with solutions based on open standards. Current information about these products can be found at: www.HPIE.com.

4

applications developed at NIST for the HP nodes. Higher-level distributed control and monitoring capabilities were implemented in the Java programming language.

The nodes communicate information between themselves using publish-subscribe and request-response messaging interaction. The communication is based on TCP/IP and uses *push* technology as the underlying messaging paradigm. Internet-based "push" technologies define a type of designer protocol that uses TCP/IP as its underlying transport mechanism to forward publish-subscribe and request-response messages. The push messaging technology allows a node to publish specific information to what is termed a *topic*. A topic is an ASCII string, for example "*node1.sensor.tanktemp*", which might represent the temperature in the coolant tank on the network. Another node would subscribe to this topic to receive updates on the temperature of the tank. The HP nodes shown in Figure 3 were programmed to provide these as well as many other application services. These services can be controlled and monitored via the Web-based Java client applet. The Java applet however does not communicate directly with the HP network nodes. The Java applet must first connect with a NIST developed TCP/IP gateway in order to request services from the HP network nodes.

The NIST-developed gateway shown in Figure 3 is a multi-threaded Microsoft Windows NT program that extends the publisher-subscriber messaging capabilities of the HP nodes to TCP/IP-based Java clients. To initiate a conversation with the gateway, a Java client must first connect to the gateway using a TCP/IP-based socket connection. After the connection phase completes, the Java client issues topic subscription requests using the TCP/IP protocol. The gateway performs publish or subscription functions on behalf of the Java client to complete the subscription process. Data received by the gateway from the HP nodes is prepended with the ASCII topic name and then forwarded directly to the appropriate Java client. The data received by the Java client from the gateway is then used to update the real-time values in the applet. The applet then graphically displays the control-loop with the updated values.

The graphical simulation of the control-loop provides the operator with an accurate picture of the process control system. To provide even greater feedback to the operator, live real-time video and audio sequences of the system in operation are provided using a Microsoft NetMeeting™ video conferencing software client. The real-time graphical simulation of the system in operation combined with the audio and video provides a unique diagnostic control mechanism on the Internet.

In addition to providing a graphical depiction of the control-loop, the applet also provides an extensive set of operator menus. These menus can be used to start and stop the HP node applications, request temperature data, actuate the valve remotely, or to request specific information about the devices in the system. Using these menus, commands can be sent from the Java client to the HP nodes via the gateway using a TCP/IP socket connection. Because these operations can be monitored and controlled remotely, operators no longer need to be in front of the machine to monitor the temperature or change system parameters in the control-loop. In fact, from any standard Web browser, the operator can log on and get up to the second information about the running process. Although the application domain in this case is concerned with remote access and control, this architecture is directly extensible to a variety of other domains.

The NIST Internet-based DMC demonstration, including the software, hardware, and standards used during the development is provided in upcoming sections. The discussion initially targets the lowest level within the NIST DMC framework; that is, what standards are used to physically connect transducers to the network nodes. This is followed by a discussion of Ethernet as a control networking medium and the software NIST developed to support open network communications. Finally, a detailed discussion of the software used to implement the distributed control applications on both the network nodes as well as in the Java client is provided.

Standardized Transducer Interfacing

At the lowest level of a distributed measurement and control system, sensors and actuators are needed to sense environmental conditions and control physical entities, respectively. Transducer interfacing refers to the process of physically and electrically connecting the transducer to the microprocessor of the smart network node. A key reason for standardizing the interface at the hardware interconnection level is due to the current compatibility problems transducer manufacturer's face when integrating their devices into multi-vendor networks [2,4]. Transducer interfacing also requires standardized software interfaces to provide application and network interoperability at the network node level. Because the network and the transducer must expose their interfaces directly to transducer applications on each node, any attempt to migrate the application, the sensor hardware, or the network node to another platform requires a time consuming and costly redesign of the application's interface to the new environment.

All prospects for interoperable, plug-and-play sensors and actuator devices are currently lost because of proprietary or unique interfaces. Transducer manufacturers must now expend a great deal of engineering effort to cover several control network vendor technologies instead of designing the device once for all networks that adhere to the standardized interfaces. If a standardized approach to interfacing the application with the network and the transducer microprocessor were available, then identical transducers and their applications would be interoperable. This would allow the selection of a network for measurement and control applications to be free from transducer compatibility constraints. Transducer application designers could then focus on their applications without being concerned about developing interfaces for every possible network or microprocessor they decide to target. Such issues have motivated the formation of the cross-industry IEEE P1451 working group to define a *networked smart transducer interface* standard.

IEEE P1451 Overview

The Committee on Sensor Technology of the Instrumentation and Measurement Society of the Institute of Electrical and Electronics Engineers (IEEE) has defined a standard for a Networked Smart Transducer. Figure 4 illustrates the placement of the proposed IEEE P1451 family of standards.
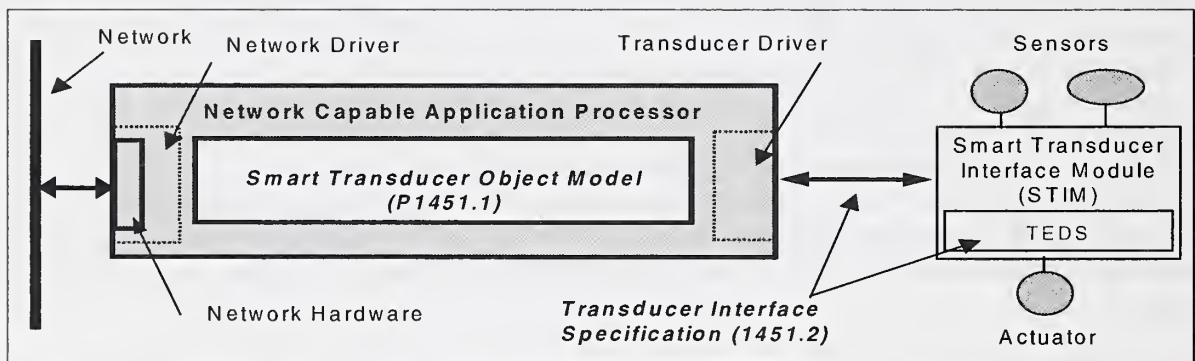
Figure 4: The IEEE P1451-based Smart Transducer including both P1451.1 and 1451.2.

The proposed standards combine a smart transducer information model (P1451.1) targeting software-based, network-independent, transducer application environments with a standardized digital interface (1451.2) and communication protocol for accessing the transducer via the microprocessor [5,6]. IEEE P1451 addresses the transducer industry's two most problematic areas: (1) the definition of a standardized digital interface between the transducer and the microprocessor, and (2) a standardization of the

application elements that impact network and transducer communication. Figure 4 illustrates the complete P1451 smart transducer standards model comprising both interfaces to the information model and the transducer interface. Figure 4 will be used in the upcoming descriptions of P1451.1 and 1451.2.

P1451.1 - Information Model

The proposed P1451.1 Draft Standard, officially known as the Network Capable Application Processor (NCAP) Information Model, provides an abstract interface description that ultimately will be transcribed into an application programming interface (API). As part of the standard, an interface definition language (IDL) has been developed that provides the standards definition in a neutral descriptive language. The IDL serves as the basis for which the standard interfaces are defined and described. Any ambiguities in the standard specification that may arise due to its abstract definition will be refereed using the IDL interface specification.

The P1451.1 information model encompasses an object-oriented definition of a network capable application processor (NCAP). An NCAP is the object-oriented embodiment of a smart networked transducer device, and is commonly called a network node. The information model includes the definition of all application-level access to network resources and transducer hardware. The object-based aspect of the information model is referred to as the Smart Transducer Object Model and is shown in the center of the NCAP in Figure 4. The Object Model definition encompasses a set of object classes, attributes, methods, and behaviors that provide a concise description of a transducer and the network to which it may connect. The model is sufficiently general to encompass a wide variety of networked transducer application services. The object model addresses the two problem areas by standardizing on the linkages between how applications interact with physical sensors and actuators in the system and how these same applications interact with the attached network.

The object model provides a framework for building highly intelligent and distributed application architectures at the network node level of the transducer device. The standard provides a network and transducer hardware neutral environment in which to develop software. In addition to the software interface that P1451.1 defines, a standardized hardware interface for digital transducer communication has also been defined.

1451.2 - Transducer to Microprocessor Interface

The IEEE has approved the 1451.2 specification[2], or the Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats specification. The 1451.2 specification provides a standardized digital interface and communication protocol that directly addresses the problem of interfacing multiple transducer connection schemes with different buses and microprocessors. The standard also includes the definition of a smart transducer interface module (STIM) and a transducer electronic data sheet (TEDS). The locations of the STIM and TEDS are shown in Figure 4.

A STIM consists of a transducer, signal conditioning, a TEDS, and necessary logic circuitry to implement the IEEE 1451.2 10-wire digital interface. The STIM includes the IEEE 1451.2 10-wire physical specification and the set of communication protocols required to access TEDS information, read sensor data, or to manipulate actuators. This specification defines the communication that occurs between the STIM and NCAP in order to obtain information. The data transfers are based on the common serial

---

[2] Notice that the "P" in the IEEE standard number has been dispensed with. The "P" represents a current Project under IEEE. An approved standard no longer retains project status.

peripheral interface (SPI), a bit-transfer protocol. The NCAP usually initiates a measurement or action by means of triggering the STIM, and the STIM responds with an acknowledgment once the requested function has completed. Each transducer in a STIM is defined by a channel. For example, a temperature sensor, pressure sensor, and airflow sensor can be combined together to form a multi-channel mass-flow sensor.

A TEDS is a scaleable and extensible electronic data sheet that allows transducer manufacturers to include information about their product such as manufacture date, version information, and calibration specifics, into a small nonvolatile memory associated with the transducer hardware. The TEDS information provides a self-identification capability for transducers that are invaluable for maintenance, diagnostics, or determining mean-time-before-failure characteristics. Prospects of this capability have generated a great deal of interest in the sensor manufacturing industry.

The NIST DMC System and IEEE P1451

Having a standardized means of connecting transducers to network nodes at the distributed control level is very important. The IEEE P1451 family of standards facilitates this standardized interconnection mechanism. Standards at this level provide transducer manufacturers, systems integrators, and network vendors with hassle-free ways of connecting their hardware to a multitude of control network topologies. Additionally, one of the most important aspects for the future is to use the P1451.1 information model in object-oriented environments to produce more sophisticated distributed architectures with networked smart transducers. These newer application architectures will yield higher levels of integration with real-time data and control networks by abstracting out the functionality to enterprise level entities. More capable processing nodes combined with ubiquitous network access at the distributed control level will ease the burden on higher level applications and network resources to integrate this critical information with enterprise information bases.

In support of the IEEE P1451 standards effort, several vendors has been developing technology to demonstrate and educate vendors at the various sensor and control technology exhibitions. One of the vendors, Hewlett-Packard (HP) initially focused on developing prototypes for the approved 1451.2 hardware interface. As part of the transducer hardware used in the NIST DMC demonstration, we employed a prototype hardware reference implementation of an IEEE 1451.2 STIM developed by HP. The HP STIM interfaces to the NCAP were used to provide the digital communication interface and TEDS information specified by the standard.

The STIM hardware interface provides the capability for the transducers used in the demonstration to be digitally connected as input directly to the NCAP. Onboard analog-to-digital conversion (ADC) circuitry converts the analog voltage levels of the temperature transducers to the digital representation used by the node application. The NCAP used was also developed at HP and was capable of directly connecting to the Ethernet. The NCAP houses the node application and was responsible for sensing the temperatures, converting the digital information into degrees Fahrenheit, and then publishing the data onto the network. A second NCAP node used a digital input/output STIM in order to provide actuation capabilities for the valve transducer.

All applications running on the nodes were developed at NIST to provide the distributed intelligence needed during the demonstration. Each node application controlled an NCAP and was responsible for operations and communications among each other. The next Section titled *Open Network Communications*, discusses the major issues in implementing a control network based on the use of Ethernet, TCP/IP, and the HP network nodes.

Open Network Communications

Network nodes must be connected to a control network in order to do useful work. Many current DMC-based control networks use proprietary hardware and software interfaces that limit the availability of data to higher-level networks and repositories. In addition, because DMC-based field networks are designed with specific application domains in mind, many companies possess several different types of device networks in order to solve different problems. Supporting multiple types of control networks in order to target very specific domains is rapidly becoming a thing of the past [8]. The need for distinct types of control networks to fulfill DMC requirements has been waning, moving more towards open and defacto standards-based data communication networks based on Ethernet technology.

Ethernet as a Control Network

The use of Ethernet as the preferred medium for DMC-based control networks is rapidly gaining in popularity because of its speed, cost-effectiveness, and the ability to leverage off-the-shelf application components to facilitate building distributed systems [1]. Changes in the IEEE 802 Ethernet specification are making it quite formidable as a network for device-level control. For example, the IEEE 802.1p standard for message prioritization or quality of service (QoS) was initially developed for streaming live video, audio, and other multimedia content. Indeed, those in the control-networking arena can leverage the real-time deterministic capabilities in the Ethernet standard. The standard effectively guarantees that messages can be delivered and or acknowledged in less than a 4-millisecond window. This range is within most tolerances for providing adequate response times for process control applications.

Opponents of using Ethernet as a control-networking medium have suggested that it will never be deterministic enough for hard real-time control network applications. Demonstrations from companies such as Hewlett-Packard have shown that very stringent time critical response times can in fact be delivered to Ethernet-based applications [16]. Recently, the Foundation FieldBus organization selected Ethernet for their control network draft specifications [9]. The FieldBus specifications define a H2 network, which is a standard for higher speed networks with data rates around 100 Mbps. High-speed Ethernet was chosen above several other competing proprietary buses.

With the advent of industrialized Ethernet and more resilient protocols for deterministic and guaranteed response times, Ethernet will be hard to beat as the network of choice in the control arena. Another major reason for the attractiveness of using Ethernet as a control medium is the fact that it has already shared widespread usage in the enterprise. NIST interest in Ethernet stems from several ongoing research projects involving the use of Ethernet as the basis for manufacturing-based distributed sensor networking. In addition, using Internet technologies to support gateway research with other control network technology has been another chief research concern [18].

The Hewlett-Packard technology used in this demonstration was one of the earliest available that supported connectivity with Ethernet networks and that provided a solid reference implementation of the IEEE 1451.2 hardware interface. Using the network nodes available from HP allowed NIST to experiment with placing sensors and actuators directly on the Ethernet, while also integrating them into key Internet-based research efforts that are going on internally at NIST [4].

Hewlett-Packard Vantera™ Communication Services[3]

By basing control network technology on an IEEE 802 based local area network physical medium, we could directly connect via a 10baseT connector the HP Ethernet network nodes to the NIST network backbone. In addition, using the TCP/IP protocol, the nodes on the control network were directly addressable, facilitating their integration into the enterprise-side of the NIST network. Although the control network for this demonstration uses Ethernet and TCP/IP as the primary network and protocol, the HP nodes communicate between themselves using a special publish-subscribe messaging protocol. The protocol, which uses TCP/IP as its underlying transport, supports the HP Vantera communication services. These communication services form the basis for the HP Vantera Information Backplane (VIB). The VIB uses TCP/IP and Ethernet to transport the publish-subscribe messages to each node in the DMC system. Vantera is the name Hewlett-Packard uses for these prototype distributed measurement devices.

The HP VIB is currently based on the Tibco Rendezvous™ publish-subscribe push technology [19,20]. Push technology is the basis for many application messaging implementations on the Internet (i.e., Marimba Castenet™, PointCast™, and BackWeb™ Technologies). The VIB's underlying Tibco messaging envelopes TCP/IP with efficient and guaranteed delivery capabilities. This technology effectively eliminates application developers from twiddling with low-level network protocols and addressing. It uses *subject*-based addressing to perform network address bindings on the LAN. Using this approach, developers need only publish or subscribe to high-level subject names in order to send or receive information. Subjects are user-defined strings such as "*node1.sensor.temperature*" [23]. Subjects are synonymous with topics when used in the context of the HP VIB. We will use the HP parlance and refer to subjects as *topics* throughout the rest of this document.

Topics are ASCII strings representing communication endpoints on the network. HP Vantera application developers use topics to allow communicating entities to publish or subscribe. This provides a very clean communication paradigm that does not bog down the developer in network specific communication details. An example of an ASCII-based topic is *node1.sensor.tanktemp*. This string represents a topic that the Java applet would subscribe to via the NIST gateway to receive the tank temperature reading from the HP node. The developer writes code to simply send or publish messages containing application specific data to these topics. Other users or clients of the topics subscribe to them, thereby receiving the data.

Although the publish and subscribe messaging paradigm is commonly used for this type of communication, the underlying technology used to implement the messaging is typically proprietary. In fact, there are currently several competing middleware messaging vendors that implement the publish and subscribe paradigm. There is industry movement to consolidate the messaging schemes into a unified approach. Middleware disparities will most likely become transparent as the messaging industry coalesces around using Internet Inter-ORB Protocol (IIOP) bridges for the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) and Microsoft's Distributed Component Object Model (DCOM).

Currently however, use of the Tibco messaging protocol means that client applications cannot communicate with a node if it does not support the HP VIB as its underlying communication system. Therefore, clients cannot communicate or subscribe to topics using ordinary TCP/IP application software interfaces without some sort of intermediary. As of this writing, Hewlett-Packard has begun implementing IEEE P1451.1 for the next generation HP Ethernet NCAP. As part of this implementation, the proprietary Tibco-based communication subsystem will be replaced with an open TCP/IP-based

---

[3] The TIBCO messaging protocol has been replaced in later versions with a multicast IP based communication capability based on DCE-OSF messaging technology

library for inter-NCAP communication. However, clients will still communicate with nodes using an intermediary process in this situation until an agreed upon TCP/IP mapping to the IEEE P1451.1 specification has been ratified.

Figure 5 illustrates a topic with its associated message on the VIB. The output of a debugging utility called *dmcrecv* is also shown in Figure 5.

```
dmcrecv -h node1.sensor.tanktemp

header={ sessionId="810624E9.DAEMON.9CB0A009C9F94.64", type=16, timestamp=Wed De
c 31 19:07:06.210526 1997 }
message={ value=72.3686, quality=0 }
```

Figure 5: Messages received after listening to the node1.sensor.tanktemp topic on the VIB.

This debugging utility captures traffic on the VIB associated with some topic name and displays its contents in the console. In this case the *dmcrecv* program is subscribing to any messages received for the *node1.sensor.tanktemp* topic. This topic represents the published coolant tank temperature on the network. The value of the temperature sensor is located in the message field of the *node1.sensor.tanktemp* topic message; in this case the tank temperature is 72.3686 degrees Fahrenheit.

Figure 6 illustrates the communication pathways between the two nodes. The data flow used directly by the control algorithm on NODE-2 is shown with two arrows in the figure.
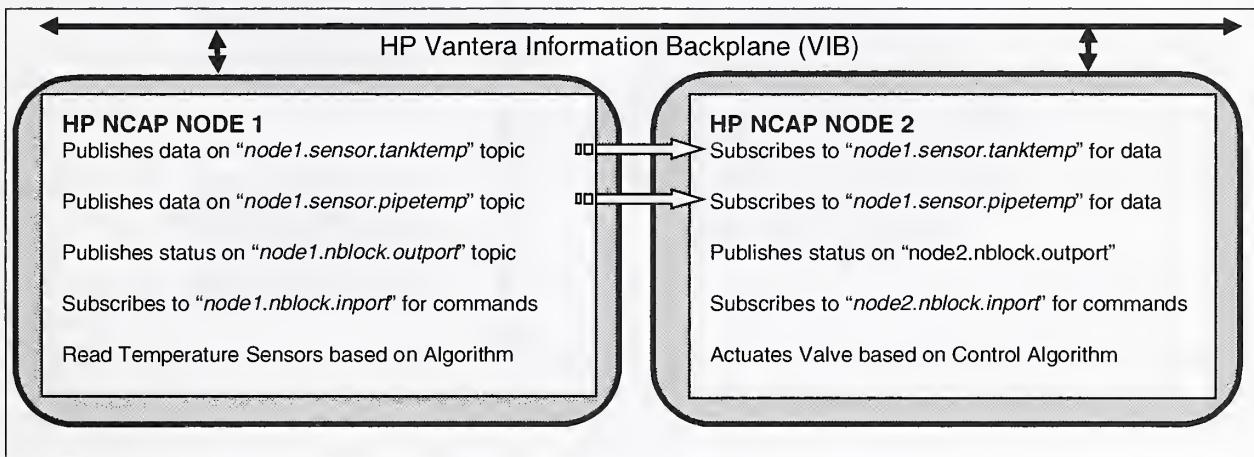
Figure 6 : The relationship in the nodes between publishing and subscribing to topics on the VIB.

All other publish or subscription topics in the nodes are indirectly being used on the VIB for client status and updating procedures. An intermediary process called a gateway provides the indirect use of the topics by the clients. The gateway was developed at NIST to provide TCP/IP client communication support to the HP network nodes. The gateway is discussed in the next section.

NIST-developed Gateway links TCP/IP clients to the HP Vantera Information Backplane

Gateways are generally unobtrusive systems that convert a protocol from one side of the network to the protocol in use by the other side. Due to the proprietary nature of the VIB communication mechanism, NIST developed a software gateway that provides TCP/IP clients with access to the VIB. In this case, the

NIST gateway acts as a publish-subscribe intermediary between the HP Vantera nodes generating the data (using push technology) and the Java client applications consuming the data (using TCP/IP socket connections). The gateway *extends* the publish-subscribe HP Vantera communication services out into the realm of the Java clients by actually performing the HP VIB functions on behalf of the client applications. The NIST developed **VGateway**, is a C language-based, Microsoft Win32 multi-threaded TCP/IP concurrent server. *VGateway* effectively performs the functions of a protocol-level application gateway. The gateway is capable of accepting requests from TCP/IP-based client applications and converting them to the appropriate communication service requests for use by HP Vantera network nodes.

Clients of *VGateway* can be written in C, C++, Java, or any other language capable of issuing standard TCP/IP-based connection setup and communication primitives. At a conceptual level, the NIST *VGateway* makes the underlying messaging paradigm used on the HP VIB transparent to the Java client. The *VGateway* allows client applications to view the HP Vantera nodes as peer entities as shown in Figure 7. In the NIST Internet-based DMC demonstration, multiple Java clients might concurrently establish communication with the HP nodes. For this reason the *VGateway* application supports multi-threading in order to provide multiple concurrent client access.
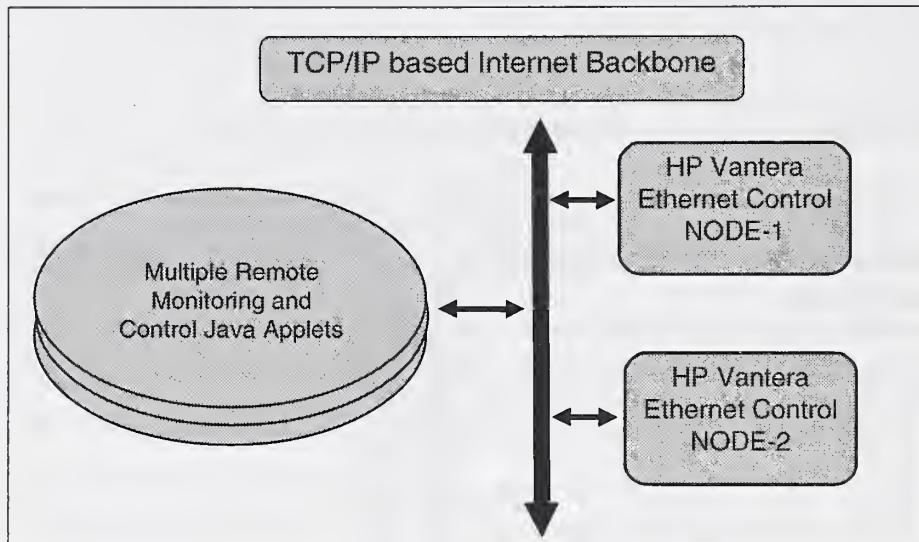
Figure 7: A conceptual view of the Java clients' interaction with the HP control nodes.

Java clients send TCP/IP-based ASCII messages to *VGateway* addressing them with specific topics. The *VGateway* receives these messages and encodes them into HP VIB communication payloads. It then multicasts the encoded message onto the VIB using the topic received from the Java client message. The node listening to the appropriate topic receives the request. The ASCII-based topical addressing scheme used between the Java clients and the HP nodes provides a common communication bridge.

In the NIST Internet-based DMC demonstration, only a few topics were needed to provide the system and the control algorithms with the necessary information to operate effectively. Three types of data values associated with each topic are sent from the nodes to *VGateway*, including:

(1) temperature sensor values in the form of floating point numbers,
(2) status messages indicating whether or not the valve was actuated, and
(3) response messages from client-server queries.

12

Table 1 summarizes the topics and messages sent by the Java client to *VGateway* and vice versa.

Topic/Address	Java Client Request Message	VGateway Response Message
vgateway.system	"vgateway.system CONREQUEST" *(client requests a connection with the gateway)*	"vgateway.system CONACCEPT" *(client was granted a connection to the gateway)*
node1.sensor.tanktemp	"node1.sensor.tanktemp SUBSCRIBE" *(client request to subscribe to the tank temperature)*	"node1.sensor.tanktemp XX.XX" *(client starts receiving XX.XX as a float value)*
node1.sensor.chilltemp	"node1.sensor.chilltemp SUBSCRIBE" *(client request to subscribe to chiller temperature)*	"node1.sensor.chilltemp XX.XX" *(client starts receiving XX.XX as a float value)*
node1/2.nblock.outport	"node1/2.nblock.outport SUBSCRIBE" *(client request to subscribe to NODE-1/2 status port)*	*(No messages returned from this request)*
node1/2.nblock.inport	"node1/2.nblock.inport GET TEDS [1,2,3,4,5]" *(client request TEDS data using messaging port)*	"node1/2.nblock.outport TEDS XXX" *(client receives XXXX as ASCII TEDS string)*
node1/2.nblock.inport	"node1/2.nblock.inport READ/WRITE/START/STOP" *(client request to read, write, start, or stop transducer)*	*(No messages returned from this request)*
node2.nblock.inport	"node2.nblock.inport GET/SET SETPOINT" *(client request setpoint function using messaging port)*	"node2.nblock.outport SETPOINT XX.XX" *(client receives XX.XX as a setpoint float value)*
node2.nblock.outport	*(no client request, result of subscribe to status port)*	"node2.nblock.outport valve ON/OFF" *(client receives ON/OFF valve status message)*
any.topic.from.above	"any.topic.from.above UNSUBSCRIBE" *(client request to be disconnected from gateway)*	"no message returned from this request" *(unsubscribe and disconnect client from gateway)*

Table 1:Topics and associated messages used in the NIST Internet-based DMC Demonstration.

As the table summarizes, the inter-node (*VGateway* is considered a node on the VIB) communication protocol is a simple yet effective set of ASCII-based topic definitions. Column 3 of Table 1 shows the responses sent back to the Java client from *VGateway* during this demonstration. In Table 1, identical requests involving both nodes are shown in column 1 topics as *"node1/2"*.

Topics act as endpoints of communication for receiving sensor data updates (i.e., node1.sensor.tanktemp is used for receiving only the coolant tank temperature), or for facilitating client-server messaging (i.e., *node1/2.nblock.inport* is used for sending command requests from the Java applet). Others are used to convey response messages received during client-server interactions. Finally, there are status-oriented messages that periodically publish state information about the running system during normal operations (i.e., using *node2.nblock.outport*).

The important difference between the topics lies in the underlying semantics used to define the endpoints. Client-server messaging is implemented as publish-subscribe using the *inport* and *outport* topic designation. *Outport* addressed topics are overloaded to return status information about the running system in addition to client-server responses. *Inport* topics are used to provide a request address when sending a client-server message. All other topics are used as subscription endpoints to provide asynchronous sensor and actuator data to the Java clients.

As an operator issues commands via the Java client interface to either node, *VGateway* acts as an intermediary by intercepting the message, decoding it, and sending the appropriate command message to the HP Vantera destined node. Likewise, when the node responds to commands, it does so by publishing the response out on the VIB. *VGateway*, acting as a communications surrogate forwards the information received by the nodes back to the clients as information arrives destined for them.

The *VGateway* design reflects the requirement that TCP/IP-based messages must be translated into HP VIB messages and vice versa. Figure 8 illustrates the various components making up the C language based NIST *VGateway*. In the NIST *VGateway*, multiple threads handle all bi-directional TCP/IP traffic with Java clients while another single thread handles all HP network node-based VIB traffic.
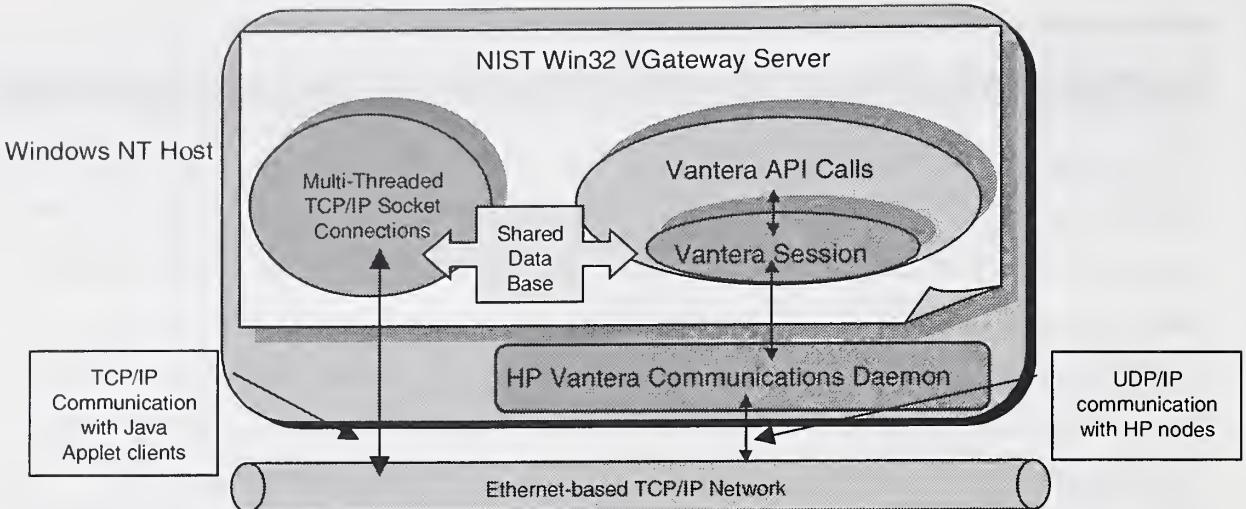
Figure 8: The NIST VGateway provides TCP/IP access to the HP Vantera Information Backplane.

Each thread uses the shared database in Figure 8 in order to address and to communicate appropriate topic-based messages to entities on both sides of the network. The multi-threaded *VGateway* design required several key software components, including:

- TCP/IP and HP Vantera Information Backplane initialization routines.
- A database of ASCII string-based topic names and TCP/IP client socket handles.
- Database routines for updating topical subscriptions and TCP/IP client connections.
- Thread functions for listening to and accepting client connections to *VGateway*.
- Per TCP/IP client-thread communications functions for translating incoming requests to the VIB.
- A threaded communications controller for translating incoming VIB responses to TCP/IP clients.

VGateway Initialization and Startup

VGateway was developed to support the well-known concurrent server model for network interaction that has been the dominant architecture for most UNIX style TCP/IP-based *daemons* [21]. However, instead of forking and executing code to cope with concurrency issues, the NIST gateway uses the thread API inherent to Win32/NT. The Win32 *VGateway main()* program creates and initializes a thread of control in the server to establish a TCP-based stream server socket. The socket in this thread listens for possible client connections on a port. The *main()* program continues by establishing communication with the HP Vantera Information Backplane by opening a *session* using the *dmcCom_openSession()* API.

Opening a session issues an announcement to all parties on the VIB that a new application exists and is capable of sending and receiving messages. When a session is opened, a connection is made to the Vantera communications daemon that is running locally on the Windows NT host shown in Figure 8. In order for *VGateway* to send and receive messages on the VIB, this session initialization with the daemon must take place. If a communications daemon is not currently running, then the open session call will start one executing on the host. Figure 8 illustrates the HP Vantera *session* connection process as *VGateway* executes on a Microsoft Windows NT host machine.

If the session opens without errors, then the main thread of execution in *VGateway* will block on any completion events that signal the end of the program. This saves CPU cycles in the Win32 console application by waiting on an event semaphore, thus releasing all CPU resources associated with the *main()* program. While the *main()* program is in a blocked state however, the other threads for accepting client connections and receiving network data are all executing freely. If *VGateway* is interrupted or the Vantera communication services start improperly, then the semaphore-based event object signals *VGateway* to clean up and terminate.

Figure 8 shows a multi-threaded Microsoft Winsock version 2.0 socket-based communication module executing within *VGateway*. This module represents the TCP/IP-based network threads in the application that are responsible for maintaining the client connections with the shared portion of the subscription database. After initial client connections with *VGateway* have been established, the client is free to send subscription messages to *VGateway* in order to begin receiving data generated from the nodes.

In order for communication between the Java client and *VGateway* to take place, the client must connect to *VGateway*. Connections with *VGateway* may be refused for a variety of reasons, including a predetermined number of users has been reached, the server could in fact be down, or *VGateway* is simply too busy to respond to the connection attempt. In general however, as long as *VGateway* is up and running, clients typically will receive a connection establishment response from the server. Upon a successful start-up, the *VGateway* program shown in Figure 9 displays the results of starting a TCP thread and opening the Vantera communication services.
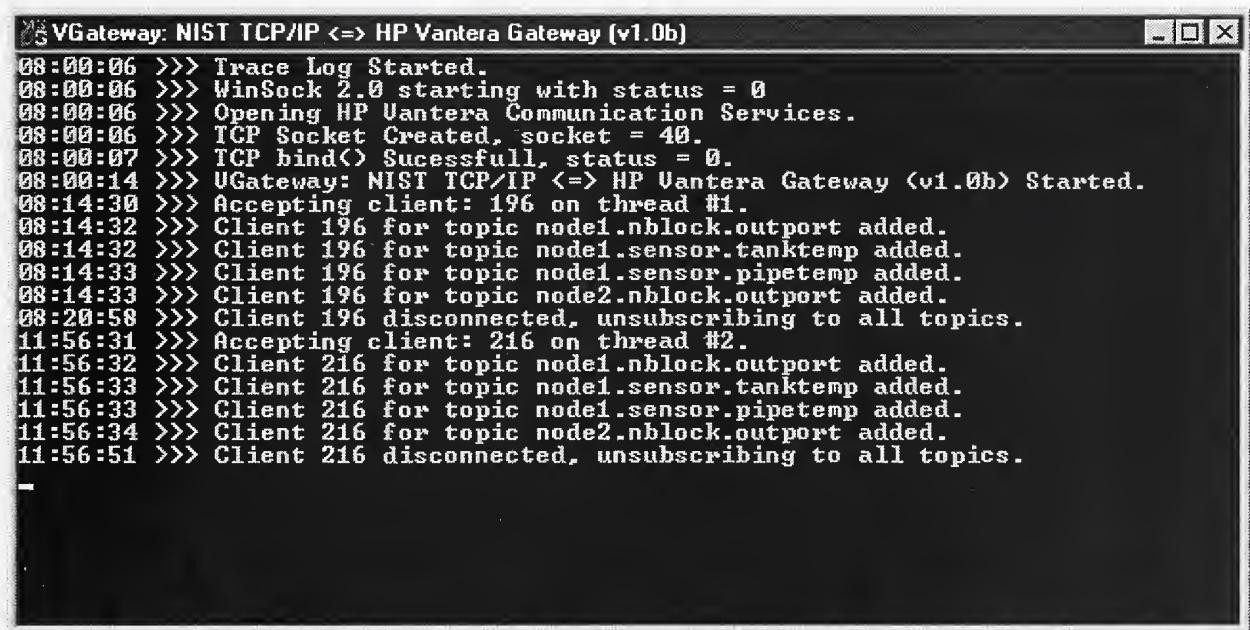
```
VGateway: NIST TCP/IP <=> HP Vantera Gateway [v1.0b]           _ □ ×
08:00:06 >>> Trace Log Started.
08:00:06 >>> WinSock 2.0 starting with status = 0
08:00:06 >>> Opening HP Vantera Communication Services.
08:00:06 >>> TCP Socket Created, socket = 40.
08:00:07 >>> TCP bind() Sucessfull, status = 0.
08:00:14 >>> VGateway: NIST TCP/IP <=> HP Vantera Gateway (v1.0b) Started.
08:14:30 >>> Accepting client: 196 on thread #1.
08:14:32 >>> Client 196 for topic node1.nblock.outport added.
08:14:32 >>> Client 196 for topic node1.sensor.tanktemp added.
08:14:33 >>> Client 196 for topic node1.sensor.pipetemp added.
08:14:33 >>> Client 196 for topic node2.nblock.outport added.
08:20:58 >>> Client 196 disconnected, unsubscribing to all topics.
11:56:31 >>> Accepting client: 216 on thread #2.
11:56:32 >>> Client 216 for topic node1.nblock.outport added.
11:56:33 >>> Client 216 for topic node1.sensor.tanktemp added.
11:56:33 >>> Client 216 for topic node1.sensor.pipetemp added.
11:56:34 >>> Client 216 for topic node2.nblock.outport added.
11:56:51 >>> Client 216 disconnected, unsubscribing to all topics.
```

Figure 9: The Win32 VGateway Console application display after client connection and topic subscription

Figure 9 illustrates the output of the Win32 console application *VGateway* during a successful start-up. The console message window in Figure 9 displays two client applications (client 196 and client 216) attaching to the gateway and subscribing to specific topics. The numbers associated with each client are the printable integer value of the socket handles obtained during the TCP/IP network connection phase. Upon client termination, *VGateway* displays the disconnection status messages and *VGateway* automatically unsubscribes to topics that the client was subscribed to.

Notice in Figure 9 how multiple connections from different clients attach to *VGateway* at the same instant. The multi-threaded server can support an unlimited number of clients; although response times to the clients will be slowed if too many connections are attempted at once. In addition, the server can accept connections from any clients in any language. In this case we are using Java; however, the C language has also been used for testing. Any language that can support TCP/IP socket communication can subscribe to topics using this generic subscription gateway.

Database of TCP Client Connections

When messages arrive at *VGateway* from the VIB network, topic and client lookups are required to disperse messages. To provide fast retrievals of subscription topics, a database consisting of several singly linked-lists is used. The topic-based linked-list contains one singly linked-list for representing the number of clients currently subscribed to a particular topic. The secondary list represents each client by storing their TCP socket handles in the list. Storing socket handles directly in the data structure to represent clients is convenient during the client updating procedure. During this procedure, all that was necessary to update each client was to iterate through the list and present to the TCP/IP *send()* function the client socket handles. Storing the socket handle as opposed to a flag or other variable allowed immediate client updating without indirect addressing. The presence of a *NULL* entry in the client list immediately signaled that no more clients have subscribed to this topic. Figure 10 illustrates the basic structure of the topic and client connections database.
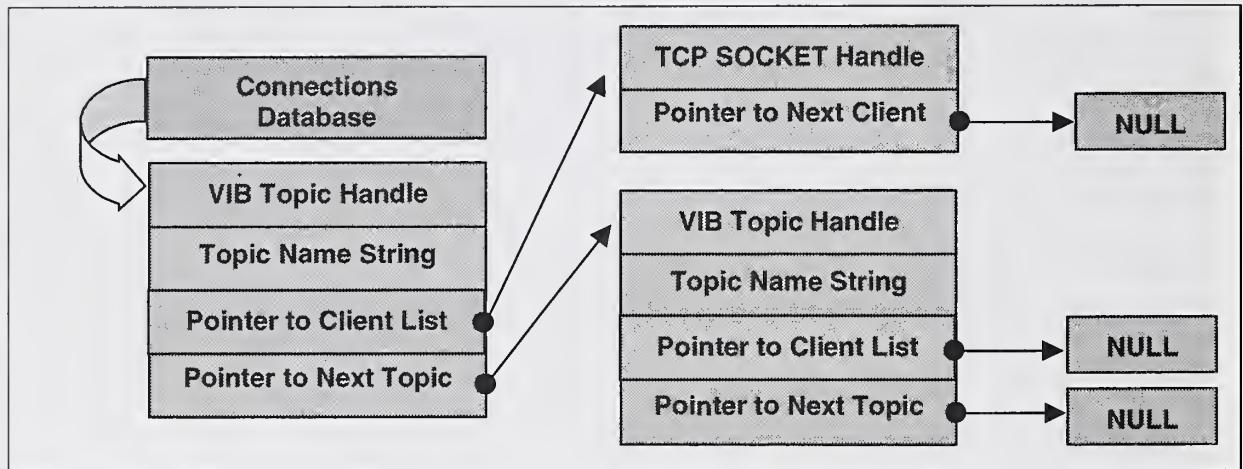
Figure 10: The Connections Database provides a dynamic linked-list of VIB topics and TCP/IP client connections.

The connections database in Figure 10 provides a simple yet efficient method of indexing topic names (the primary key). If the topic exists, the pointer to the client list consisting of TCP/IP socket handles can be quickly traversed in order to update each client in the list. Self-maintaining linked-list update routines eliminated holes in the structure to allow maximum efficiency during searches. Other fields in the database include a VIB Topic Handle that contains the VIB specific identifier for the topic on the network. This variable is used when the initial subscription process takes place. The *VGateway* subscribes to a topic on the VIB on behalf of the client. A VIB specific handle is returned upon successful subscription. The handle is used later in an unsubscribe call to disassociate the subscription binding on the VIB session. An ASCII string topic name in the database is used for string compares during message receipt routines in *VGateway*. Lastly, a pointer to the next topic name in the linked-list is maintained in the connections database. This is the standard linked-list method of addressing the next item in the list. In

16

a singly linked list such as the connections database, the last pointer in each list will be set to a NULL value indicating the end of list. This provides a highly efficient dynamic data structure.

A Centralized, Thread-Safe VIB Communications Controller

Another key network design goal in *VGateway* was to provide an ability to *listen* on behalf of the client for incoming messages from the HP VIB side of the network. Broadcast data values published at regular intervals as part of the nodes normally executing control algorithm are received at *VGateway* using a centralized thread-safe callback mechanism. Upon receipt of a message, *VGateway* forwards the message to the awaiting client using the callback. The gateway *listens* to certain topics on behalf of clients in the same way the HP node applications respond to incoming network messages from the VIB. A C language callback function is registered with the Vantera runtime system during the *VGateway* initialization. This function is responsible for handling all network related incoming message traffic from the VIB destined for the TCP/IP clients.

The pseudo-code in Figure 11 illustrates the callback function *subscribeCallback()* used by *VGateway*. A C language library for the VIB was provided for the Win32 environment that could be used to register the callback in the same way as the node applications. This allowed VIB specific messages to be received by *VGateway* in a Windows NT environment. VIB related messages received by *VGateway* were translated into TCP/IP-based ASCII messages before sending them to the Java client.

```
/*
 * If this callback fired, then an incoming HP VIB message is ready for processing by the gateway.
 */

subscribeCallback()
{
        lock access to multi-threaded user data and linked-list
        if (incoming message topic (i.e.,"node1.sensor.tanktemp") is in the database)
        {
                retrieve linked-list element referring to this topic
                determine what the HP VIB payload type refers to (float, string, integer, etc.)
                build a TCP/IP-based Ascii response packet containing the converted payload
                use the retrieved topic element to access the client TCP/IP socket list

                while (there are clients currently subscribed to this topic)
                {
                        send client the updated TCP/IP message data
                        increment client pointer to the next client in the list
                }
        }
        release the mutual exclusion lock on the callback function
}
```

Figure 11: VGateway processing and routing of all incoming message traffic from the VIB to the appropriate clients.

As the pseudo-code in Figure 11 illustrates, the callback is used in a multi-threaded environment. Mutual exclusion locks are needed when the function attempts to access user-defined data such as the connections database. For performance reasons, instead of registering one callback per session with the Vantera

runtime, a callback per topic subscription algorithm could be implemented. For the NIST DMC demonstration project a single callback to handle all traffic from the VIB was sufficient.

When *VGateway* receives a message on the VIB for a particular topic, the *subscribeCallback()* function is called by the Vantera runtime system linked with *VGateway*. At that time, the database is locked for access by the thread invoking the callback function. The database is then searched on the primary key for a topic match. If a match is found, the topic element is retrieved from the database. Using the matched topic element from the database the current subscription client list is extracted.

The payload information in the HP VIB message is then decoded and converted into an ASCII equivalent for building a TCP/IP message. The ASCII message that was built from the HP VIB payload information is sent to each client by iterating through the client list for that topic. The iteration mechanism terminates when a *NULL* condition for the next client is found. Upon terminating the loop the database is once again unlocked and the function *subscribeCallback()* returns. If a topic subscription was not found, then the database is unlocked and the function returns immediately.

The locking mechanism shown in Figure 11 is used to protect the user-defined data during the callbacks execution and is provided by the developer. In addition to the programmer supplied thread synchronization primitives, there are mechanisms provided by the HP Vantera runtime system library to keep the communications routines safe. The mechanisms to keep the executing callback thread-safe are used throughout the HP Vantera runtime system. Therefore, the only thread safe issues the developers must contend with are the self-imposed ones due to possible multi-threaded user access to user-defined data structures or variables in the application.

Distributed Control Applications

Applications for control systems can take several forms. Most notable are segregated or individual control, centralized, and distributed control [17]. Segregated control is not relevant to this discussion as it functions in a standalone non-networked environment. Centralized topologies typically use a master-slave application interaction scheme. In this scheme, a master controller delegates to slave nodes on the network. Centralized control has by far been the most commonly implemented in the past. Distributed control however is fast becoming the norm for designing control systems. Lower cost microprocessors, higher-speed networks, open communications, and higher level languages and development environments make them more cost-effective to implement. Implementing a standardized distributed control application framework is the key focus of this Section. This standardized application framework is possible by using high-level languages, off-the-shelf development environments, defacto network interfaces, and IEEE-based transducer integration techniques.

In the NIST DMC demonstration project, the control is based on the distributed model. In addition, within the distributed model of control, two forms of distributed interaction are presented, namely: (1) autonomous low-level control algorithms implemented in the C language on transducer-based network nodes, and (2) high-level monitoring and control using the Java programming language and a Web browser. The application of standards to both of these areas will be highlighted in this Section. In addition, the partitioning of the applications responsibility is an important area of study. Using a distributed model for application development and system partitioning is rapidly becoming the norm. The traditional as well as the newly emerging distributed approach to node application design is discussed in the upcoming Section titled *Node-based Distributed Applications*. Internet-based, high-level distributed application design is discussed in the later Section on *Web-based Distributed Monitoring and Control*.

Node-based Distributed Applications

The IEEE P1451.1 draft standard provides an abstract application programming interface for NCAP transducer developers to write portable software on the nodes. The P1451.1 object-oriented framework for developing NCAP-based application objects lends itself to supporting sophisticated and highly distributed systems. This is a fundamentally different paradigm for application development then those found in traditional DMC system development partitioning.

Traditional centralized controller-based DMC system development makes use of a master-slave method for systems design. In the master-slave design, a node or device is designated as the master controller of all subordinate devices. Figure 12 shows the master delegating system processing by polling slave devices for information.
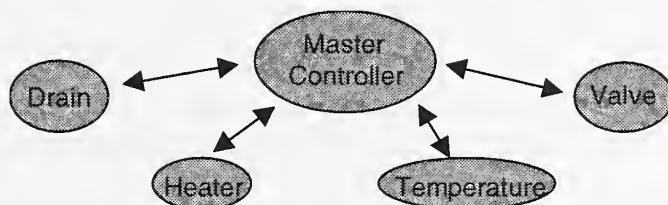

Figure 12: Master-slave control in a DMC environment.

As the master controller gains information, it sends out control signals to the devices needing attention. The application graph shown in Figure 12 represents a typical control scheme that provides thermostatic

control of water located in a tank. The controller in the master-slave model becomes a bottleneck should information transfer to higher levels become necessary.

As self-describing information and data access is not directly supported by the nodes, all data request, identification, and application interaction with the nodes must be proxied through the controller. Should the controller go down or have other catastrophic problems, then the entire system is in jeopardy. Integrating this type of architecture with higher level information models clearly becomes overloaded and convoluted. Efficient system designs as well as future standards integration issues demand a more scalable distributed design alternative to the master-slave model. This kind of system design is rapidly disappearing as distributed system frameworks become the norm to solve these types of problems.

Distributed design alternatives are coming of age in part because of a new breed of control systems category called *soft control* [22]. Soft control systems replace hardware controllers and interfaces with software-based control. This is seen in programmable logic controller (PLC) systems that are based on specialized hardware, use inflexible architectures, and perform data exchange via hardware interfaces. Soft control-based systems use general purpose scalable hardware that facilitates data communication using higher-level network interfaces. The software-based characteristics of soft control facilitate the move to distributed system architectures.

Solving the thermostatic control-loop in Figure 13 using a distributed system design decentralizes processing to individual smart transducer-based networked nodes. In the distributed design, each node has autonomous control over the device it is controlling. Based on receipt of a variety of input, each node makes a decision and carries out the appropriate action. Using a messaging scheme, the nodes communicate requests/responses to their peers or perform publish/subscribe activities to relay information on the network. An application graph illustrating the distributed design of the thermostatic control scheme is shown in Figure 13.
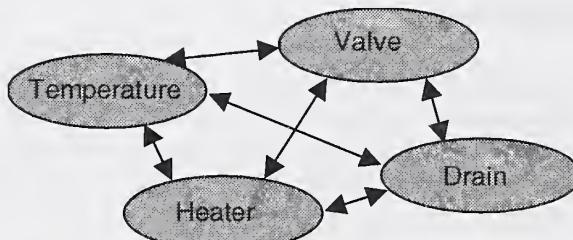

Figure 13: Solving the thermostatic control-loop problem using a distributed design.

Adding distributed intelligence to nodes provides many benefits to the application designers as well as system integrators. Self-identifying features of the transducers and the nodes provide maintenance engineers with immediate feedback from failures in the network when replacement parts are required. Failure and recovery schemes in distributed networks can be achieved gracefully. As the system learns of an outage, it can respond, possibly by starting a redundant node. Master-slave systems typically require that all the intelligence be contained within the controller. The controller must provide for all contingencies due to the slave nodes' limited intelligence, preventing them from making autonomous decisions.

Designing systems in a distributed and fault-tolerant manner becomes a more natural design alternative to master-slave as the network nodes themselves become more powerful and intelligent and the application models become more complex. As the marriage between a microprocessor and network processor becomes solidified and more cost effective, more capable sensor network appliances will become prevalent.

Programming environments that support these systems will undoubtedly be object-oriented by design and support sophisticated modern operating system services. The resulting distributed applications will be extremely intelligent and highly integrated within the organizations information infrastructure and the manufacturing process.

IEEE P1451.1 Node Execution Model

As part of the NIST DMC demonstration project, distributed node applications for the Hewlett-Packard Vantera platform were developed using a flexible framework based on a finite state automaton execution model. When developing node applications in the C language for the HP Vantera environment, the developer structures the application to fit into specific states used by the node during its execution lifecycle. All node applications execute within the node's real-time operating system (RTOS) environment (Wind River Systems *VxWorks*™). The operating system moves the application through various states depending on certain criteria. The node application defines specific execution code paths based on this state model. The application code is placed into the appropriate state by the runtime framework of the operating system. While in each state, the NIST DMC demonstration node application program executes code paths relative to that state.

The states that the node application passes through are loosely defined by the Draft IEEE P1451.1 block object state definition. The states mimic those defined by the abstract IEEE model when describing state and behavior of NCAP block objects as they pass in and out of existence [11]. The application can go through a series of states during its lifecycle such as CREATING, RUNNING, and DELETING. These states, when projected onto a node application, are implemented as C language functions with the same name. For example, the RUNNING state appears in the node application source code as a C API function called *running()*. The RTOS uses this function as an entry point to call when it determines that the application should move into the RUNNING state.

The CREATING state provides an area where all data structures, publication registrations, and subscription handles are initialized. Any runtime mutexes (mutual exclusion primitives), or application specific user-defined data structures are initialized to base values here as well. Functional code segments are typically not executed in the CREATING state as the environment that the application will run in has not yet been completely setup and initialized.

The RUNNING state contains most of the application-specific functional code. The real-time operating system loader moves the node application into the RUNNING state after initialization from the CREATING state has been performed. At that point the application is running the algorithms that the transducer application developer has defined. The RUNNING state provides developers with a location for placing continuously executing application-specific code. After the application reaches the RUNNING state, it will continually execute the algorithm that defines it purpose until the application has either been stopped via external services or the node has been rebooted.

The DELETING state consists of freeing memory associated with the node application's user-defined and system-specific data structures. The system-specific data structures are those used by the application to pass information to callbacks during runtime invocation. These are stack-based parameters inherited by the system during the invocation process and are used by the application to determine, for instance, the mapping between the topic and network message. Other application data structures include large global structures used to house centralized application-specific data during algorithm execution. Typical data elements found in this structure include: (**1**) status flags associated with receiving data, (**2**) storage areas for sensor data values, (**3**) variable storage for the next command to be executed by the RUNNING state

algorithm, (4) subscription and publication handles, and (5) array storage for the network message received. All of these storage locations need to be released upon application completion.

Event-driven Node Application Architecture

In addition to the application code found in the RUNNING state, the node developer must provide additional functions that asynchronously respond to messages addressed to the node from the network, or as internal events are generated. HP node application developers can add functionality to the application by defining special functions that include user-defined code. These functions are independent of the state-based functions and take the form of *callbacks*. Callbacks in the HP Vantera system are C language functions that are called by the HP Vantera node runtime system when various events occur in the system.

In general, both node applications used in the NIST DMC demonstration project have been designed using asynchronous event-driven callbacks. The majority of the callbacks used in the NIST application environment are based on events such as the receipt of network messages arriving at the node or by timer events being fired by an internal application defined scheduler. As topic-based network messages arrive at the node, the HP Vantera runtime system will call a specific callback that has been registered for the topic. The node application must have previously subscribed to the topic in order to receive topic messages. As part of the subscription process, an address to a user-defined C language callback function is given to the runtime system. When a message arrives on the VIB for a specific topic, the HP Vantera runtime system calls the callback. This style of invoking user-defined code based on callbacks is pervasive in the event-driven software simulation and control areas. Figure 14 illustrates the use of the HP Vantera runtime system by the node application. The callback functions are invoked by the HP Vantera runtime system when network messages arrive or timer events expire. The application callback design shown in Figure 14 is the one used by NODE-2.
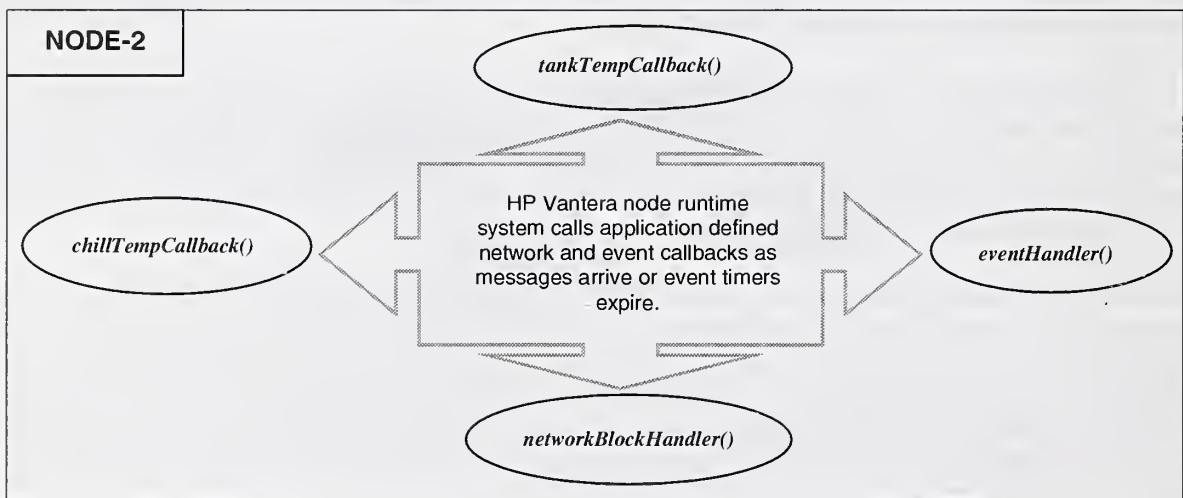
Figure 14: Using the HP Vantera node runtime system to asynchronously call user-defined functions in NODE-2.

When external events such as network message receipt or event expiration occurs, one of the four callbacks defined for NODE-2 in Figure 14 will be called. The four callbacks: *tankTempCallback()*, *chillTempCallback()*, *eventHandler()*, and *networkBlockHandler()*, combine to form the basic architecture of the nodes' application. NODE-2 receives external temperature data from the network in order to make a determination to actuate a valve. NODE-2 uses *tankTempCallback()* and *chillTempCallback()* to asynchronously receive tank and chiller temperatures that are being published to the network from the temperature transducers on NODE-1. At user-specified intervals (changeable through the Java client applet interface), the temperature readings from the two sensors located in the coolant tank and the chiller

22

are sampled and published on the network from NODE-1. NODE-2 subscribed to these temperatures via a call to the HP Vantera *dmcCom_subscribe()* C API. After NODE-2 moves into the RUNNING state, the application is ready to receive callbacks from the receipt of either tank or chiller temperature network messages from the VIB.

NODE-2 uses the temperature values during its control-loop algorithm execution cycle to determine whether or not to turn on or off the valve. The control-loop algorithm running on NODE-2 is driven by an internally scheduled event mechanism. Just as the sampling rate for reading the temperature sensors on NODE-1 is controlled by a timer event callback, so to is the event scheduling mechanism for applying the control-loop algorithm on NODE-2 (user selectable in the Java front-end). NODE-2 defaults to schedule an event to occur every 2 seconds in order to execute the distributed control algorithm. The callback *eventHandler()*, is called as a result of event timer expiration. This callback implements the user-defined function to drive the default control-loop algorithm.

The control-loop algorithm uses a specified setpoint temperature as the coolant tank temperature goal. NODE-2 maintains this temperature by checking every 2 seconds to see if valid temperature data has been received from the network. If so, the valid tank temperature is checked against the currently defined setpoint (set through the Java front-end). If the current tank temperature is above the setpoint, then the actuation value flag is set to true; otherwise, the flag is set to false. The actuation function is then called that will write the correct value to the IEEE 1451.2 compatible valve. The correct value being ON (true) if the setpoint has been exceeded or OFF (false) if the setpoint has been maintained.

The final callback, *networkBlockHandler()*, provides a generic interface to handle client-server command requests from Java applet clients. That is, two special topics (*node1/2.nblock.inport*) were set up so that clients could send messages to each node in order to effect some change or action on the node. Both nodes subscribe to this topic in order to receive messages from the clients via *VGateway*. The network handler callback provides a centralized area to receive, parse, and act on specific commands sent from the clients. This was necessary in order to provide the capability to change the characteristics of the running node application in real-time.

The *networkBlockHandler()* callback also provides a way for client applications to retrieve important real-time information from the nodes pertaining to the IEEE P1451 standard definition. For example, retrieving IEEE-defined TEDS information from each node such as the transducer manufacturer, hardware description, as well as reading individual channels from the transducer could all be done using the network callback mechanism. Other client-server based services such as starting and stopping the node application, changing the sampling periods, or changing the setpoint for the NODE-2 algorithm could all be done using this interface.

The *networkBlockHandler()* callback receives the command message, parses the packet, and determines what if any action the application is to perform. In order for the network block callback to carry out the actual command however, it needs to call another runtime function called *dmcApp_setSignal()* to signal the node application that a specific command requires processing. Recall that once the node application has successfully moved to the RUNNING state, a block of code repetitively executes some common transducer application code. We designed this repetitive function to respond only to asynchronous software signals generated from within the application. The RUNNING state algorithm waits forever until a software signal generated by the application occurs. As the RUNNING state wait loop catches application generated signals (representing commands from the external world), it enters a case statement that determines what command is being requested. The command selector used in the case statement was previously obtained by parsing the message received during the *networkBlockHandler()* invocation.

Several different types of commands may be executed while in the RUNNING state's case statement. Some of these include starting and stopping the node application (this consists of scheduling event generation), setting the sampling rate (how often to fire events), writing ON/OFF data values to the actuator transducer channel (to manually turn on or off the valve), retrieving various string portions of the TEDS fields (i.e., manufacturers and calibration data), and finally, retrieving the setpoint (this was useful for clients to obtain the initial setpoint upon startup). After the command is processed, the RUNNING state forever loop continues to wait. The waiting was handled in the signal loop using a HP Vantera C API *dmcApp_requestWait()* call that facilitated the wait-loop semantics.

A start command uses the HP Vantera C API call *dmcSched_addUTC()* to schedule events. A time data structure is initialized with the default sampling period or one obtained through user intervention from the client. Stopping the node consist of terminating all event generation in the node. Terminating the event generation in the node began by querying the runtime environment using the *dmcSched_contains(id)* call to see if an outstanding event of this type was running. If so, event generation terminated by calling the *dmcSched_remove(id)* function with the specific event identifier as a parameter to the call.

Changing the sampling period affects how often the application's control-loop algorithm executes. When a request to change the sampling period was received, the effect on the application was as if a stop command was issued to the application. After the event generator was stopped, a new event was scheduled using the newly selected sampling period. Therefore how often the node control-loop algorithm executes was a direct function of setting the sampling period from within the Java client.

Separate functions for writing to the actuator channel, getting the TEDS, and getting the setpoint were also created for these types of commands. Writing to the actuator channel was provided by the user-defined *actuateChannel()* function located in the *running()* function case statement. This function calls the HP Vantera C API *dmcXdcr_writeUint32()* to send a binary data value out to the digital transducer channel. The function sends a "one" for ON and a "zero" for OFF when directing the valve actuation mechanism. The *getTEDS()* user-defined function was used to obtain the TEDS self-identification strings from the transducer. The *getTEDS()* function called the HP Vantera C API *dmcXdcr_get()* call to obtain the actual TEDS values from the transducer IEEE 1451.2 hardware. Specific parameters were provided to the call to define what TEDS value the user requested. All HP Vantera functions for communicating with the sensors or actuators in this demonstration used the IEEE 1451.2 standard implementation.

The user-defined function *getSetPoint()* was used to get the current setpoint value from the node. This function accessed a common application data structure for the current setpoint and built a response message with the appropriate value. All client-server responses sent from the nodes would use the same mechanism for sending the requested information back to the client. That is, the special topics (*node1/2.nblock.outport*) were set up so that outgoing messages in response to client requests could be directed to *VGateway*. *VGateway*, listening on behalf of the clients collects any node responses (using the network callback technique) and iterates through each connected client to deliver the response. Clients send requests to *inport* topics and listen for replies on *outport* topics. The clients are responsible for parsing messages received on *outport* topics to determine how to use the information.

The NODE-1 application structure is very similar to NODE-2. Subtle differences exist in the types of commands the network block callback function processes. Recall that NODE-2 used the default sampling period to determine when to run the control algorithm internally. NODE-1 also uses a sampling period to determine how often the temperature sensor data should be obtained. Therefore, one additional user-defined command supported by NODE-1 provides the ability to *readChannel()*. This function calls the HP Vantera C API *readFloat()* function to obtain the degrees Kelvin reading from the sensor channel and

convert it to degrees Fahrenheit before publishing the data to the network. Using the sampling period to drive this activity, the temperature data is obtained and published to the VIB.

Overall, the application designs of the two nodes are quite similar. The main difference being that NODE-1 is the primary producer (publisher) of data while NODE-2 is the principal consumer (subscriber) of the networked data. NODE-2 uses the information to mediate the temperature in the coolant tank by controlling the valve actuation process. All nodes produce status information and all nodes are capable of fielding client-server requests from the Java clients.

The two nodes execute autonomously, however, it becomes clear that they are controlled and manipulated through external clients. The Java client represents the high-level application decoupling of distributed measurement and control functions. That is, although the network nodes execute the device-level distributed algorithms associated with the control-loop, the Java applet also drives specific high-level application parameters in the control-loop. This Web-based decoupling of the application is a powerful concept and is the basis for the Section titled *Web-based Distributed Monitoring and Control*.

Web-based Distributed Monitoring and Control

Web-based strategies for monitoring and controlling distributed measurement and control applications are fast becoming an important area of commercialization. As the need for more intelligence at the network node increases, so do high-level methods for accessing, distributing, and controlling the information generated at these levels. Sophisticated information delivery tools are needed to present, access, and distribute the information generated from the DMC systems to the enterprise. Information delivery to the enterprise has seen a dramatic shift in recent years from standalone dedicated databases, tools, and applications to vastly de-centralized client-server based enterprise-wide systems.

Most notable during this shift is the concept of using the World Wide Web for storage and delivery of content coupled with Web browser technology for displaying the information. Network-aware software such as Java has played a significant role in ushering in these new information delivery mechanisms. The NIST Internet-based DMC demonstration project illustrates how these Internet information delivery technologies can be used to provide ubiquitous access and control to DMC systems on the plant floor from around the globe.

In order to implement this multi-level control strategy and application framework, we chose to develop at the Internet level a remote monitoring and control application using the Java programming language. With inherent Internet capabilities built-in, and the write once execute anywhere portability benefits, the Java language is well suited to provide the graphical user interface and networking requirements for this demonstration. Figure 15 illustrates the NIST developed Java applet for remote monitoring and control using the Microsoft Internet Explorer browser 4.01. This application is available on the Internet at: http://motion.aptd.nist.gov/P1451/ISADemo.htm. This demonstration was featured at the IEEE P1451 booth at the International Society for Measurement and Control (ISA) Tech97 Exhibition in Anaheim, California in October of 1997. An updated version of this demonstration was shown at the ISA Expo/98 exhibition in Houston, Texas in October of 1998.

The ISA Expo/98 show highlights advanced technology in the measurement and control industry. In order to demonstrate the application remotely using the Internet, a notebook computer was used at the show to dial into the NIST Intranet using a 28K modem to establish a point-to-point protocol (PPP) connection. Once connected, the Microsoft Internet Explorer Web browser was executed containing the Web page for the NIST real-time coolant control Java applet client shown in Figure 15.

The animated graphical Java applet simulated the conditions of the control-loop in real-time using data obtained from the network. Audio and video were also available from NIST in real-time using Microsoft NetMeeting™ 2.1 over the 28Kbps dial-up modem connection to the Internet. The video sequences showed the inline flow meter turning as a result of the 3-way valve actuation process. The remote real-time audio and video illustrates an exciting new capability for remote diagnostic and monitoring on the Internet.
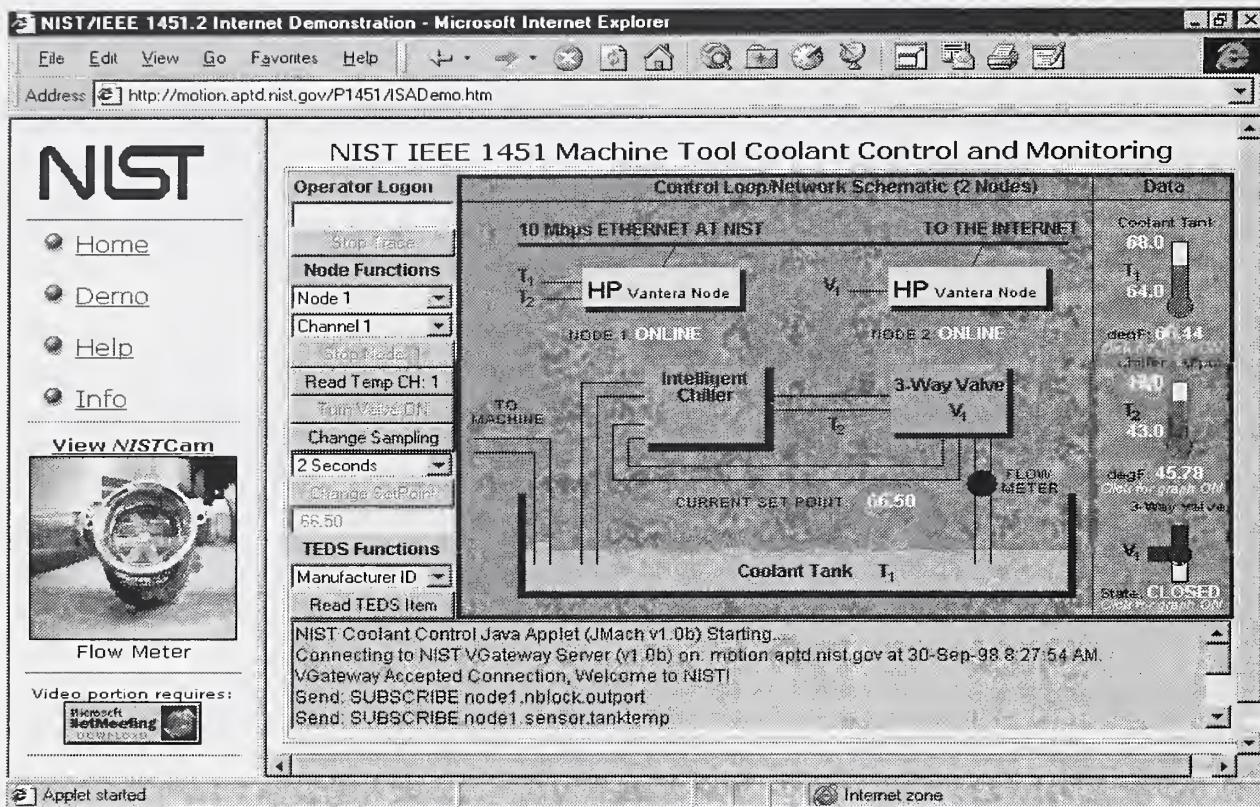

Figure 15: The NIST Java-based remote monitoring and control applet.

The demonstration graphically depicts a closed-loop control system that regulates the temperature of machine tool coolant. This system allows the operator to control the temperature of the coolant to a consistent temperature for long running machine tool processes. In addition, the applet allows the operator to control and query the control system for key state information. Menu-based operator commands were added to illustrate the variety of useful information that the IEEE P1451 standardization process has defined. These commands are important for remote control and monitoring of diagnostics, maintenance, and recovery mechanisms. Besides illustrating a real world application in use on the Internet, the demonstration provided a reference platform for the integration and use of the NIST DMC standardization framework.

As shown in Figure 15, the demonstration consists of two prototype HP Vantera network nodes that provide the framework for the distributed control application in the demonstration. NODE-1 senses the temperature in the machine tool coolant tank and in the intelligent chiller output line (i.e., temperature sensor inputs T_1 and T_2). NODE-1 publishes (at user selectable sampling periods) the specific temperature sensor values to the network using topics. NODE-2 subscribes to the topics corresponding to the two

temperature sensors thereby receiving the values. NODE-2 then uses this information to determine whether or not to actuate a 3-way valve (i.e., the valve output V_1) on or off. NODE-2 controls the actuation algorithm of the valve by using the distributed information it receives from the other node. Both nodes publish internal state information for maintaining synchronization as well as providing status information to the network for interested clients. The Java client having subscribed to these status related topics via the gateway, uses this information to update its graphical control-loop.

NIST Client-Server based Java Implementation

In this demonstration, the Java applet represents the client-side of the client-server communications model. The NIST software gateway provides the key server-side capabilities allowing Java clients to connect, subscribe, and communicate various command requests to the server. Java can readily be used to construct powerful client-server distributed application architectures as the core Java specification includes a TCP/IP networking API. The NIST Java applet uses the core *java.net* package to implement the client-server application distribution. This allows the Java applet client to connect to the NIST TCP/IP software gateway to access the nodes on the DMC network. The NIST Java applet was developed as a series of five object-oriented classes. Each of the five classes represents a logical aspect of the applet, including:

1. General applet class: (*JMach*) Supports a generic multi-threaded applet environment.
2. Control-loop animation class: (*ControlLoop*) Provides an animation of the distributed control-loop.
3. Thermometer class: (*Temp*) provides a graphical display of temperature sensor data.
4. Valve class: (*Valve*) Provides a graphical representation of the valve actuator state.
5. Network Class: (*NetThread*) Supports the TCP/IP-based network client communications.

The general applet class (*JMach*) provides the application framework for the rest of the applet. It is the main driver for the application and is responsible for instantiating other objects in the system. The *JMach* applet is graphically intensive, therefore the majority of the Java source code was developed specifically in support of the applets graphical user interface capabilities. The menu selection area and the control-loop display simulation encompassed three of the five classes and comprise the majority of the source code. Graphical capabilities of the Java applet include displaying:

*   thermometers with accurate real-time temperatures that rise and fall based on the data received,
*   valve movement with accurate state changes,
*   true movement of coolant through the piping in relation to the actuation process,
*   digital updates of the node status, setpoint, and temperature readings, and
*   trending graphs of the temperature readings and the valve state for comparison in real-time.

The specific implementation of the graphical capabilities will not be discussed in this paper as they utilize low-level graphics programming techniques from the abstract windowing toolkit (AWT) of the Java core specification. The graphical classes will indirectly be discussed in terms of how the application supplied any necessary data elements to them for textual or graphical display.

JMach – a Java-based real-time monitoring and control applet

The *JMach* class encapsulates the standard Java (JDK 1.0) applet functionality by providing an *init()*, *start()*, *stop()*, *destroy()*, and *paint()* method. The *JMach* applet also implements the runnable interface. The *JMach* applet class is multi-threaded, and contains a *run()* method that allows background processing to continue while the main applet executes.

When a user accesses the URL referring to the NIST demonstration, the Java applet is downloaded to the users local browser from the Web site hosting the HTML page. The browser begins initializing the classes of the applet by executing the Java applet within the browser environment [23,24]. As the Java virtual machine (JVM) begins the process of executing a Java applet's bytecode[4], it will first call the applet's *init()* method entry point. The applets' *init()* method is responsible for initializing data structures it will use in the application. In addition, it will setup any graphical user interface components making up the display. All interactive commands that can be issued from the Java GUI are handled using an *action()* method. This is a centralized event-based routine that handles all possible user interface events that could occur once the applet is executing[5].

Next, the applet's *start()* method will be called automatically by the JVM. Should the browser operator navigate to a different page, the JVM will call the applets *stop()* method because the applet is no longer being displayed and hence does not need to continue execution. Likewise, when the page containing the applet is once again reached, the JVM will call the applets *start()* method once again. The *JMach* applet class uses this standard behavior to facilitate efficient use of networking resources.

The *JMach* applet implements the runnable interface, meaning the applet is multi-threaded. As a side-effect of the *start()* method being invoked by the JVM, the applet's *run()* method is automatically invoked as well. The *run()* method implements the only method associated with the applet's threaded interface. This method is a forever loop in the applet that simply sleeps for 200 milliseconds before forcing an update of its graphical display. The updating process repaints the display with any new changes to the applet that have taken place since the last sleep was called. These changes include the graphical updates needed to simulate movement of coolant within the system. As graphical updates are performed, new data values are constantly being received from the network. Implementing the applet as a multithreaded class allows the GUI updating objects to execute freely while allowing the *JMach* applet to respond to user input or field network related events at the same time.

NetThread – a threaded TCP/IP network communication class

The Java applet obtains real-time data from the network using the NIST *VGateway* server over a TCP/IP based Ethernet. To do this effectively, the "client" (the Java applet) needs to connect with *VGateway* using TCP/IP sockets. Java provides core TCP/IP networking support as well as a well-defined distributed model for Internet/Intranet-based interaction. Once connected, the client "subscribes" to various topics of interest. In this application setting, the topics represent the data channels for the two temperatures (chiller and tank temp), and an additional subscription channel for the valve actuation status.

The network class *NetThread*, combined the Java core TCP/IP networking capability into a threaded class to support the connection setup and communication with *VGateway*. The threaded network class communicates with the server using a predefined ASCII-based protocol in order to facilitate command and control processing. The Java applet acts as a thin client in this architecture relying on the server to route all information from the nodes back to them. The routed information from the server was delivered to the client using two distinct mechanisms: synchronous responses received from the nodes based on commands requested, and asynchronous responses received based on the subscriptions. The client using *VGateway* however received all responses.

---

[4] Java bytecode can be thought of as a portable object file format that is interpreted and executed by the browsers' JVM.
[5] The NIST Java applet is based on Sun Microsystems Java Development Kit (JDK™) 1.0. Newer JDK's such as 1.1 support completely revamped methods for handling events in the user interface of Java programs. At the time of development however, there were no Java-aware browsers that support the newer JDK's.

The Java client may make synchronous requests using the menus from the user interface. A request such as, "*Get me the temperature reading from the sensor on channel 2 of NODE-1*", would be initiated by the client using the user-defined Java method *sendRequest()*. The *sendRequest()* method in the *NetThread* class would package the request as an ASCII-based TCP/IP request and send it to *VGateway* addressed to the "*node1.nblock.inport*" topic. The node would receive this message from *VGateway* and then respond by publishing the answer back out to the network. *VGateway*, listening to the appropriate topic would receive the nodes response and forward it on to the Java client. By this time the Java client has issued a *getResponse()* method to synchronously block until the message is received from *VGateway*.

This scenario is illustrated in Figure 16 where *VGateway* is functioning as the publish-subscribe forwarding service on behalf of the Java client applet. After *VGateway* receives a request from the Java applet, it publishes the request using publish-subscribe onto the VIB. The node subscribed to that topic receives the message and returns the answer in a publish as well. *VGateway*, having earlier subscribed to inbound topics from the nodes receives the message from the VIB and forwards it along to the awaiting network thread in the Java applet. The messaging between *VGateway* and the nodes uses asynchronous-based publish-subscribe messaging to implement the client-server interaction from the Java client. As *VGateway* listens to client subscriptions on the VIB, it fields the response messages being published from the nodes and sends them back to the Java client using a synchronous response.
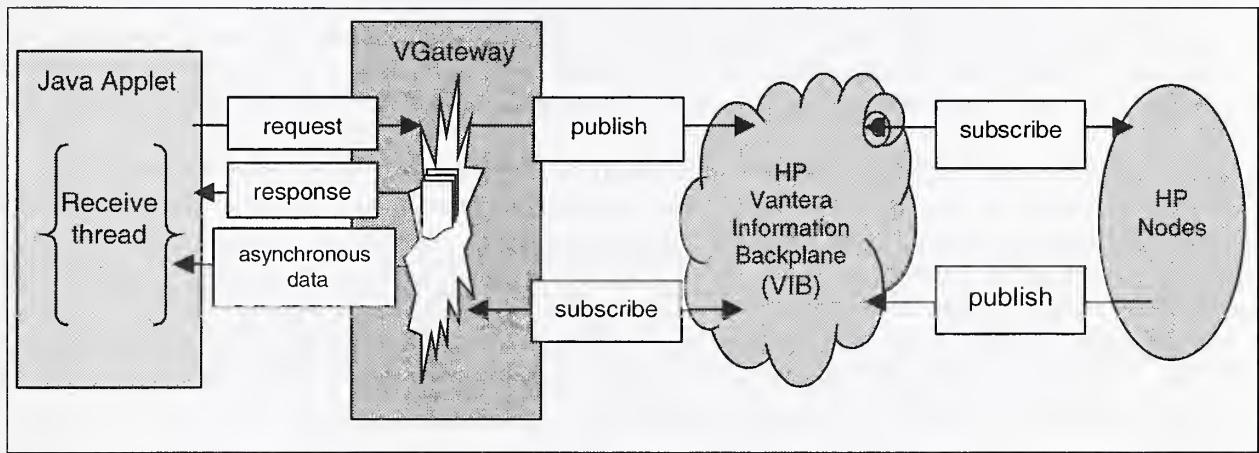

Figure 16: Satisfying Java client request/response messaging using publish/subscribe control network interaction.

Another form of communication between the client and server shown in Figure 16 uses asynchronous based messaging. In this method the Java client performs a synchronous request to *VGateway* for topic subscriptions. *VGateway* registers these subscriptions, performs the necessary actions, and listens to those topics for any matching messages. When *VGateway* obtains information on that topic, it forwards the information on to the Java client. However, the Java client does not know when or how much information will be coming into its network connection. In this sense, the communication is asynchronous and not demand-driven by the client.

To provide an asynchronous messaging capability, the Java client must implement a separate network thread. This thread receives data in the background (asynchronously), while the Java applet continues with other tasks such as responding to the user interface, drawing graphics (painting/updating), or issuing other request-response network commands from the menus. The methods used by the *NetThread* class are briefly described in Figure 17. In general the API's support the synchronous client-server communication paradigm with ancillary connection setup and tear-down methods. All background or asynchronous communications are handled centrally in the *NetThread* class *run()* method.

NetThread Class

```
openSocket();        - opens the TCP/IP socket w/VGateway
connectRequest();    - request a connection to VGateway
sendRequest();       - send an ASCII-based request message
getResponse();       - synchronous wait for a VGateway response
unSubscribeAll();    - unsubscribe to all current topics
closeIO();           - force close notification w/VGateway
```
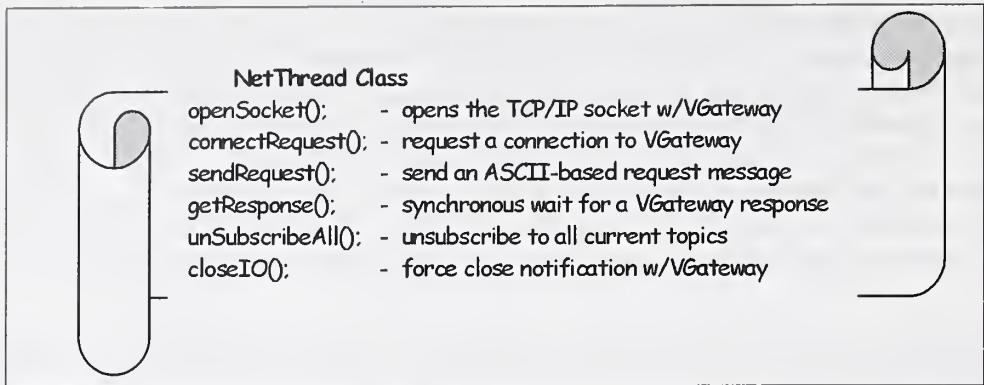
Figure 17: The NetThread class provides API's for connection, subscription, and request-response messaging.

When the Java applet begins executing, it immediately (without user intervention) performs connection setup with *VGateway* by sending a CONREQUEST message to the server executing on the Web site hosted machine[6]. This is done by calling the *connectRequest()* method in the *NetThread* class. If *VGateway* accepts the connection from the applet, then the connection phase completes with the server sending a CONACCEPT response message back to the awaiting applet. At that point the applet begins sending several back-to-back subscription request messages (SUBSCRIBE) to the server using the *sendRequest()* method. The server records these topics in a connections database containing both subscription information and currently connected clients. *VGateway* uses this information to determine whom to "forward" published data to based on the subscription information in the connections database.

The *NetThread* class effectively becomes structured as a *monitor*[7] by using inherent synchronization primitives found in the Java language. Java provides a programmer friendly method of thread synchronization by simply adding the keyword *synchronized* to a method or variable. For example, the *sendRequest()* method shown in Figure 18 uses the synchronized keyword to effectively control multi-threaded access to the socket connection during user interface activities. This provides an effective way of avoiding deadlock conditions until all locks or references to the method or variable have been exhausted. In the *sendRequest()* case, only one thread of control could use the socket connection and its associated data structures until the previous thread had completed.

The multi-threaded *NetThread* class also provides a *run()* method. The *run()* method is called when the *NetThread* class executes a *start()*. The *run()* method processes incoming messages from the TCP/IP connection with *VGateway*. The pseudo-code for the *run()* method is shown in Figure 18.

In normal operation, the multi-threaded Java applet simply receives real-time information from the server and updates its graphical control-loop based on this information. The applet uses a separate thread for providing all network related services. This includes the receipt of asynchronous data coming into the applet's network thread as a result of *VGateway* forwarding published data to the applet. The applet receives this information asynchronously as the Java applet has pre-subscribed to various topics during its initialization. Once the initialization period has passed, data will start to flow into the applet directly from the nodes as forwarded by the gateway.

---

[6] Security restrictions in Java only allow applets to connect using the network to other entities if the entity resides on the same machine that the applet class files were downloaded from; otherwise the applet needs to be digitally signed. This is known as the "Java Must Phone Home" scenario as presented in the O'Reilly book on Java [23].

[7] Monitors in computer science are a synchronization tool used to avoid deadlock conflicts when two or more entities are trying to access a given resource at the same instant; the concept was first developed by C. A. R. Hoare [25].

30

```
/*
 * Synchronized method for sending command messages to the VGateway server.
 */

public synchronized sendRequest( Ascii command string )
{
        copy the Ascii command into a socket-based buffer
        make sure the socket is valid and has been opened correctly
        send the buffer out over the socket connection
        return, releasing the thread's synchronization primitive
}

/*
 * Background thread for processing incoming TCP/IP messages from the VGateway server.
 */

public void run()
{
        while ( true )
        {
                receive incoming messages from tcp/ip connection socket
                parse topics (i.e.,"node1.sensor.tanktemp") for appropriate behaviour
                update data elements that correspond to topic (temperature, status, etc.)
        }
}
```

Figure 18: Pseudo-code for the NetThread run() method

Our distributed application design and development discussion has focused on two important areas. These areas are differentiated by their relative placement in the framework. At the lowest level, the transducer network node programming model is a distributed model. The nodes are programmed in higher level languages using TCP/IP based protocols. At the higher level, distributed application access is provided by a Java applet and is used to effect some change in the lower-level DMC system or to query operational information about the running system. Together, the distributed software provides an Internet-based DMC system that executes autonomously at the transducer while being accessed, monitored, and controlled from the Web.

Conclusion

In this paper, the NIST Internet-based DMC demonstration project that spans all levels of the DMC system has been described. This multi-level data communication ability is a viable one because of the use of standardized network, protocols, and transducer interfaces. Common Internet-based software technology was used to provide for the ease of data migration between the various communication pathways.

DMC-based Ethernet applications will continue to rapidly emerge as the demand for cost effective, high bandwidth, and intelligent transducer applications increases. Many corporate networking strategies already rely on Ethernet and TCP/IP in their data-centric Intranets. Therefore, merging Internet technology with distributed control, supervisory control and data acquisition (SCADA), PLC, distributed I/O, and smart field devices will require considerably less effort. In addition, deterministic response times, reservation-based protocols, and real-time protocol initiatives all provide compelling reasons for companies to use Ethernet in their future industrial or control networking strategies. Finally, point-to-point wiring harnesses and grids are becoming prohibitively expensive to install and maintain. A re-evaluation of how to connect field devices to controllers must be made. By using a network-based approach, the wiring problems found in traditional point-to-point systems are eliminated. Future expansion and installation of additional transducers can easily be accommodated by using a scaleable networked-based distributed approach.

It is important to realize that manufacturing software is ambitiously integrating the types of messaging middleware found in the HP Vantera Information Backplane. These types of middleware will continue to evolve and change the way software entities communicate and pass data between application components. It is now becoming clear that TCP/IP variants and Ethernet will dominate as the protocol and physical medium of choice for sensor-based networking. Providing applications such as the NIST *VGateway* server to perform the application communication services at the TCP/IP protocol level are reasonably straight-forward to accomplish in a short time-frame, using high-level languages and sophisticated feature-rich development environments.

Spanning the enterprise with a common set of general-purpose network hardware, software, and protocols will provide a seamless, multi-level network backbone capable of communicating heterogeneous information to all levels in an organization easily, efficiently, and in a time-frame that can provide critical decision-support success. Real-time production data or machine utilization numbers can easily be obtained and used by management at all levels. Value-added applications and databases can manipulate the data at various levels for presentation to enterprise personnel. Remote monitoring and control of critical aspects of the manufacturing process can now be done anywhere using off-the-shelf software.

As smart devices become more and more capable (due to lower cost microprocessors, interfaces, etc), the requirements system designers demand from them will increase dramatically. This means specifications that consider future deployment of *smarter* field devices, applications and technology need to surface now. This is what the IEEE P1451 specification purports to provide. By providing standards at the device level that allow capable devices to connect to ubiquitous networks, a broader range of applications for smart transducers can evolve. In addition, by linking a more capable, lower-level control network with a common communication medium such as TCP/IP and Ethernet, a greater flexibility of control and data accessibility can be achieved.

References

[1] ARC Corporation. *Automation Strategies: Ethernet Gaining Ground in Control Networks*. Automation Research Corporation. Dedham, Massachusetts. October, 1997. pp. 3-4.

[2] Johnson, N. Robert. *Building Plug-and-Play Networked Smart Transducers*. Sensors Magazine. Peterborough, New Hampshire. October, 1997.

[3] Madan, Pradip. *Device Bus? Field Bus? Or Sensor Bus?..... Is this Segmentation Obsolete?*. Sensors Magazine. Peterborough, New Hampshire. October, 1997.

[4] Schneeman, Richard D., Lee, Kang B. NIST Interagency Report 5711. *A Standardized Approach for Transducer Interfacing: Implementing IEEE-P1451 Smart Transducer Interface Draft Standards*. National Technical Information Service, 5285 Port Royal Road, Springfield, Virginia. 22161. October, 1996.

[5] IEEE. *IEEE P1451.1 D1.83, Draft Standard for a Smart Transducer Interface for Sensors and Actuators Network Capable Application Processor (NCAP) Information Model*. Institute of Electrical and Electronics Engineers, Inc., New York, New York. December 16, 1996.

[6] IEEE. *IEEE P1451.2 D2.01, Draft Standard for a Smart Transducer Interface for Sensors and Actuators Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*. Institute of Electrical and Electronics Engineers, Inc., New York, New York. August 2, 1996.

[7] ARC Corporation. *Rapid Pace of Network Technology Could Flatten Fieldbus*. Automation Research Corporation. Dedham, Massachusetts. August, 1997. pp. 36-38.

[8] ARC Corporation. *Automation Strategies: Ethernet-based Control Network Strategies*. Automation Research Corporation. Dedham, Massachusetts October, 1997. pp. 1-31.

[9] Baranski, Barbara and Strothman, Jim. *Ethernet picked for FieldBus's H2 Networks*. InTech Magazine. ISA Services Inc. Research Triangle Park, North Carolina. April, 1998.

[10] Sheble, Nicholas. *11th Annual Control Market Outlook: Networking steps to the front of the class*. InTech Magazine. ISA Services Inc. Research Triangle Park, North Carolina. January, 1998.

[11] Winkler, Robert. *Connecting Sensors to Ethernet: A Cost Effective, Open Systems Solution*, Proceedings of the 1996 Sensors Expo - Philadelphia, Pennsylvania. October 22-24, 1996. pp. 373-6.

[12] Ahmad, Pasha and Hertler, Curt. *Improve enterprise performance using network communications*. InTech Magazine. ISA Services Inc. Research Triangle Park, North Carolina. January, 1998.

[13] Jennyc, S. Kenn. *Linking Enterprise Business Systems to the Factory Floor*. Hewlett-Packard Journal. Hewlett-Packard, Company. Palo Alto, California. February, 1998.

[14] Madan, Pradip. *Infranets, Intranets, and the Internet*. Sensors Magazine. Peterborough, New Hampshire. March 1997.

[15] Warrior, Jay. *Smart Sensor Networks of the Future*. Sensors Magazine. Peterborough, New Hampshire. March 1997.

[16] Eidson, C. John., and Cole, Wesley. *Closed Loop Control Using Ethernet™ as the Fieldbus*. ISA Services Inc. Research Triangle Park, North Carolina. October, 1997.

[17] Bryan, L. A., Bryan, E. A. *Programmable Controllers: Theory and Implementation*. Industrial Text Company, Inc. Atlanta, Georgia. 1988.

[18] Schneeman, Richard D., Lee, Kang B. NIST Interagency Report 6136. *Multi-Network Access to IEEE P1451 Smart Sensor Information Using World Wide Web Technology*. National Technical Information Service, 5285 Port Royal Road, Springfield, Virginia. 22161. July, 1997.

[19] Hewlett-Packard. *HP Vantera™ Communication Services - Concepts Guide*. Hewlett-Packard Publication No. E2726-90001. Palo Alto, California. April, 1997.

[20] Tibco, Inc. TIB/Rendezvous™ - White Paper. Palo Alto, California. 1997.

[21] Stevens, Richard. *UNIX Network Programming*. Prentice Hall Software Series. Princeton, New Jersey. 1990.

[22] Dieraur, P., Peter. *PLCs giving way to smart control*. InTech Magazine. ISA Services Inc. Research Triangle Park, NC. March, 1998.

[23] O'Reilly and Associates. *Java Nutshell and Examples*. O'Reilly Press. Sabastopol, California. 1998.

[24] Sun Microsystems. Advanced Java Programming. Sun Microsystems Press. Palo Alto, California. 1997.

[25] Hoare, C. A. R. *Principles of Computer Science Theory*. Prentice Hall Press. Princeton, New Jersey. 1962.